Summer 2021

# Deep Reinforcement Learning With Accelerated Reward Function Technique For Robotics Task Planning

Shifabanu Mohammed Rafiq Shaikh
*San Jose State University*

# DEEP REINFORCEMENT LEARNING WITH ACCELERATED REWARD FUNCTION TECHNIQUE FOR ROBOTICS TASK PLANNING

A Thesis

Presented to

The Faculty of the Department of Electrical Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Shifa Shaikh

August 2021

The Designated Thesis Committee Approves the Thesis Titled

DEEP REINFORCEMENT LEARNING WITH ACCELERATED REWARD FUNCTION TECHNIQUE FOR ROBOTICS TASK PLANNING

by

Shifa Shaikh

APPROVED FOR THE DEPARTMENT OF ELECTRICAL ENGINEERING

SAN JOSÉ STATE UNIVERSITY

August 2021

| | |
|---|---|
| Harry Li, Ph.D. | Department of Computer Engineering |
| Robert Morelos-Zaragoza, Ph.D. | Department of Electrical Engineering |
| Birsen Sirkeci, Ph.D. | Department of Electrical Engineering |

ABSTRACT

DEEP REINFORCEMENT LEARNING WITH ACCELERATED REWARD FUNCTION TECHNIQUE FOR ROBOTICS TASK PLANNING

by Shifa Shaikh

The objectives of this research are to investigate, design, develop, and implement a DRL-based reward function enhanced robotics technique to achieve ”human-like” operations for a six DoF robot. Technical challenges for robotics applications arise nowadays with a need to achieve human-like intelligence. DRL can formulate human-like intelligence by maximizing a long-term reward function. However, the design of states, actions, policies, and reward functions having continuous values turns out to be quite tricky. Lastly, to reach a target position, control motion behavior without calculating inverse kinematics, and update state space for actions and policy optimizations are also challenging. Inspired by Google Deepmind’s published research, we propose utilizing of SAC technique to solve the above challenges. In this research, we introduce gain factors in the baseline reward function for enhanced performance of the robotics operations. We integrated and developed algorithms and software tools on a state-of-the-art Unity 3D platform for mathematical modeling and simulations. The experiments consisting of reaching the object are conducted in a simulated environment a few times. We plot training curves and conclude that our reward function is better than the baseline reward function. We note an improvement of about 259.5% over the baseline reward function.

## ACKNOWLEDGMENTS

I would like to thank my professor, Harry Li, for his continuous guidance, support, motivation, and patience throughout my research. I was lucky to have him as my advisor and mentor.

I would also like to thank my parents, friends, and well-wishers for their emotional support. Lastly, I would like to express sincere gratitude to the CTI One Corporation team, especially to Mr. Yusuke Yakuwa, Chee Vang, and Nisarg Vadher, for supporting me and helping me to get through this process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

| | |
|---|---|
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| AR | Augmented Reality |
| AUC | Area Under Curve |
| CNN | Convolutional Neural Network |
| CPU | Central Processing Unit |
| CSV | Comma-separated Values |
| CV | Computer Vision |
| DDPG | Deep Deterministic Policy Gradient |
| DNN | Deep Neural Network |
| DoF | Degree of Freedom |
| DRL | Deep Reinforcement Learning |
| DQN | Deep Q-Network |
| GPU | Graphics Processing Unit |
| GUI | Graphical User Interface |
| IP | Internet Protocol |
| LAN | Local Area Network |
| MAN | Metropolitan Area Network |
| MDP | Markov's Decision Process |
| ML | Machine Learning |
| MODEM | Modulator-Demodulator |
| NM | No Machine |
| ONNX | Open Neural Network Exchange |
| PPO | Proximal Policy Optimization |
| RL | Reinforcement Learning |
| ROS | Robot Operating System |
| SAC | Soft Actor-Critic |
| UI | User Interface |
| URDF | Universal Robot Description Format |
| VR | Virtual Reality |
| YAML | YAML Ain't Markup Language |
| 2D | Two Dimensional |
| 3D | Three Dimensional |

# 1 INTRODUCTION

Robotic systems with an increasing number of sensors, actuators and computationally intensive tasks have to interact in a complex 3D environment to achieve human-like intelligence [1]. Technical challenges arise due to a need for human-like operations for robotics systems, and there arises a need to solve the challenges per the idea of Industry 4.0 [2]. To drive the system and complete the tasks with human-like accuracy, we need an efficient algorithm. DRL comes into the picture to teach machines to learn lessons independently without human intervention. One of the goals of Industry 4.0 is to have a smart factory where robots happen to have abilities to perform tasks without human intervention and even learn novel tasks effortlessly [3]. Learning a new job is easier for humans but complex for machines. A few complex applications in industries like picking and placing objects needs human-like intelligence and accuracy. Therefore, filling this gap between the robot's performance and the need for human-like intelligence in sequential robotics tasks is an important research topic [4].

## 1.1 Background

DRL enables a system to learn various tasks in robotic operations with minimal to no human interference [5]. These tasks are realized by exploring and exploiting the surroundings and gaining rewards [6]. Although it sounds simpler to train an actor using the exploration process, but it becomes complex with an increase in complexity of the tasks. The actor must filter the experience and quickly generate the best and timely actions in a fast-changing environment [7]. The author claims that an addition of a human-like effect is not straightforward for the robots. Therefore, there is a need for a successful policy search method for robotics applications. Also, there should be an adequate amount of space to store the learned experiences. Several research papers provide surveys based on improving learning process using policy search methods, especially in our area of interest [8]. Therefore, we apply an optimum policy-based algorithm, SAC, to gain

long-term rewards [9]. Our research achieves human-like effect for industrial robots by maximizing a reward function or optimizing the policy search algorithm.

Robotics applications require accurate perception to control tasks. Piece-picking robotic systems have not yet reached the market due to lack of one of the factors, such as perception technologies. Through perception knowledge, a robot can handle control tasks quickly without any supervisor on the job. The tasks require an accurate location of an object using CV from cameras embedded in a robotic system. CNN architecture uses a sliding window to build object proposals for localization of things [10]. We propose integrating CV on DRL application for better perception and localization of the object handling tasks.

### 1.2 Problem Statement

For many robotics applications, there is a need to accomplish human-like accuracy in the tasks. Simple tasks can be easily hard-programmed. Nevertheless, as complexity increases, control and execution become arduous for the programmer. A robotics application, for example a cooking robot, has to perform operations such as reaching a target object after accurate localization in a 3D world coordinate system, avoiding damaging it while grabbing or handling, and other complex tasks. Human-like accuracy means to successfully handle all of these tasks without making mistakes. For example, the robot might not reach a correct object by evaluating wrong distance and reach a wrong place. An enabled person can complete these tasks carefully without any errors when assigned the same tasks. Therefore, we expect our robots to be as accurate as humans while handling such tasks.

As complexity of a task changes, a programmer might find it difficult to predict all the states and control actions. Therefore, to avoid coding of every possible state and action of a robot for a particular task, DRL comes in handy. The issue even arises more when the robot has continuous actions. Due to continuity in the application, a programmer cannot

codify the infinite possibilities. DRL is a renowned field to ease the problem. A robot learns the tasks and takes actions based upon its state. According to its action and transition state, the robot receives a reward in terms of feedback. This process of DRL of collecting rewards, assists robots in learning the tasks on their own. Thus, maximization of the reward function becomes a necessity for any learning process. Integrated cooking tasks contain sub-tasks like reaching, grabbing, and placing. We focus on maximization of reward function for reaching the object and leave enhancement of the reward function of other sub-tasks for future research.

### 1.3 Objectives and Proposed Work

The main objective of this research is to achieve human-like operations for robotics applications. We implement DRL, a state-of-the-art technology, to achieve the goal. An essential advantage of using DRL is that it avoids hard-coded solutions for robotics applications. It also solves the need for human-like performance by training the robots to do pre-defined sequential tasks.

To increase complexity in the tasks, we choose a robotic arm with 6 DoF movements. For straightforward interpretation and improvements, we take a task of reaching an object. The robotic movement can have unlimited states and actions through motor operations. Therefore, we cannot apply a discrete DRL algorithm for such a complex application. Instead, we choose Google DeepMind team's novel SAC algorithm specially designed for continuous control applications [9]. The DRL algorithm tends to train the robot by accumulating rewards at each state. We mainly focus on long-term rewards instead of the short-term rewards.

To maximize long-term reward, we modify an existing reward function for our application. We use Google team's and Raju K's reward function as our baseline reward function [9], [11]. We expect to get accelerated training and a higher cumulative long-term reward by an introduction of gain factors. For training and modifications, we

use Unity 3D simulation platform. Unity has the latest package for the development of DRL applications. We even show the physical system setup and future work for other researchers.

# 2 STATE-OF-THE-ART TECHNOLOGY

## 2.1 DRL Technique for Human-like Operations

RL came into existence as a significant development in deep learning. Sequential tasks in any control process can be hard-coded such as, if a system is in state-1, take action-2, receives feedback, or if in state-2, take action 5 and receives another feedback. These hard-coded programs or sequential tasks perform well for completely known applications. However, the complexity increases with increased dimensionality of the tasks [12]. A robot can have multiple states and stages. An actual state of a robot would consist of arm movement, including joint angles or velocity and Cartesian coordinates. For a 6 DoF robot, the state dimensions can go up to $2 * (6 + 3) = 18$ state dimensions and six continuous actions. To process such higher dimensional vectors, the processing of sequential tasks can be tedious. For real-time dynamic applications, the programmers are unable to predict all situations a system can encounter. Additionally, a search and plan decision making algorithm (Brute Force Search Algorithms) also fails because if the system has not programmed/encountered a strange situation, then the control is not able to reach the goal state [13]. The decisions of such systems are biased towards specific actions. To make learning process smooth, we can use RL. Thus, for dynamically changing systems, RL solves the problem. RL can bring ease to the programmers while coding and gives equal importance to each state and action.

RL makes learner learn sequential task/s using the idea of MDP [14]. RL consists of an environment and an agent. The exact difference between an agent and an environment is that anything that is beyond the control of the agent is the environment [15]. The author forms an environment dividing it into a set of states and actions to control a system. The main idea is to get the best performance from the system by deciding on an optimal state and action of the system. Policies come into existence to have an intended effect on the system's output by taking appropriate action in a state and receives a feedback, or a scalar

reward [15]. The learner learns gradually from optimal policies through various iterations and accumulates rewards at the end. Then, the performance measurement is according to the maximum rewards accumulation concerning the policy. For example, a pick and place function using a mobile manipulator, Kuka miiwa, solved a path planning problem by maximizing the reward signal [16]. The authors argue that the pick and place of the object is a mature process when considered in a structured environment but vice versa when there are high variability/unstructured scenarios. Therefore, we focus on the idea of maximizing as well as accelerating the process of accumulating reward function.

Making a learner (agent) learn on the actual device is often a tedious method which is online learning [17], [18]. Simulators make the learning process easy by mimicking the environment and situations. The use of simulators before moving onto real-physical implementation, generally called offline learning, makes the learning process faster and safer [19]. For example, a robot might require thousands of cycles to explore, learn and exploit specific tasks, making it time-efficient. With the learning process still going on, the robot can damage itself if faced with some mistakes. Offline learning becomes advantageous during the exploration and exploitation process.

Many algorithms tend to have discrete action spaces because of the benefits of the exploration and convergence issues [20], [21], [22]. However, it is often not sufficient for realistic environments to have a policy that repeatedly gives a discrete action, especially in robotics. The motor torque should be explicitly continuous to make the robot move and handle tasks correctly. However, there are a few approaches that can discretize the continuous action spaces [23]. The limitation of such approaches is when there are multiple motors and torque to control. In this scenario, the number of discrete actions elevates exponentially [24]. Nowadays, tasks having continuous action space has become an important topic for researchers to optimize and build a converging algorithm [5], [9], [23], [25]. We take a simple application of a robot having six different

axes of rotation and/or bends with a task of successfully reaching a target object placed at an arbitrary position.

Model-free methods are of two types: value-based learning and policy-based learning. Value-based learning learns an action by choosing the best action in a particular state. At the same time, policy-based learning learns stochastic policy by mapping a state to an action. Following policy gradient, solving a DRL problem needs a slight modification of parameters [26]. The policies derived in the value functions are from the discrete number of Q-values, but they cannot be applied directly to the continuous action space. Lillicrap et al. use a complex continuous action space to combine it with DQN and deterministic policy gradient to obtain DDPG [5], [23]. With this consideration, we first decided to apply the DDPG algorithm to maximize action-value function and optimize the process at every step [23]. Though, after coming across a novel SAC algorithm, we finally switched our baseline algorithm [9]. The authors proved that the DDPG algorithm is difficult to stabilize and fine-tune hyper-parameters of the environment. Moreover, eventually, it becomes difficult to extend DDPG to high-dimensional complex tasks. Also, the SAC algorithm out-performs PPO without any further tuning of hyper-parameters [25]. As a base of our research, we carry out experimentatio n on a complex task such as reaching the target and optimizing the policy by modifying the action-value function.

### 2.2 AR/VR

With an increase in sophistication of the DRL algorithms, primitive environments and agents become less valuable, elevating the demand for a universal platform. A universal platform here means an inter-active, scalable, transferable platform. A general platform makes learning process simple, increase visual complexity, easily distinguishable physics application. Unity Technologies team claims that the Unity 3D engine platform suffice these needs. This section discusses the Unity 3D engine as a DRL simulation platform, generally used for game development. Unity has recently built a package for DRL known

as Unity ML-Agents Toolkit. According to the authors, the Unity engine successfully deploys learning agents as it has a flexible UI, interactive and sophisticated physics [27].

Unity has an interactive and an easy-to-use GUI. Fig. 1 shows the GUI of Unity Hub when we launch Unity Hub to start any project. It shows our ongoing discussion project *RobotArmMLAgentUnity* and Unity version. It even informs a user about the last update/edit in the project.

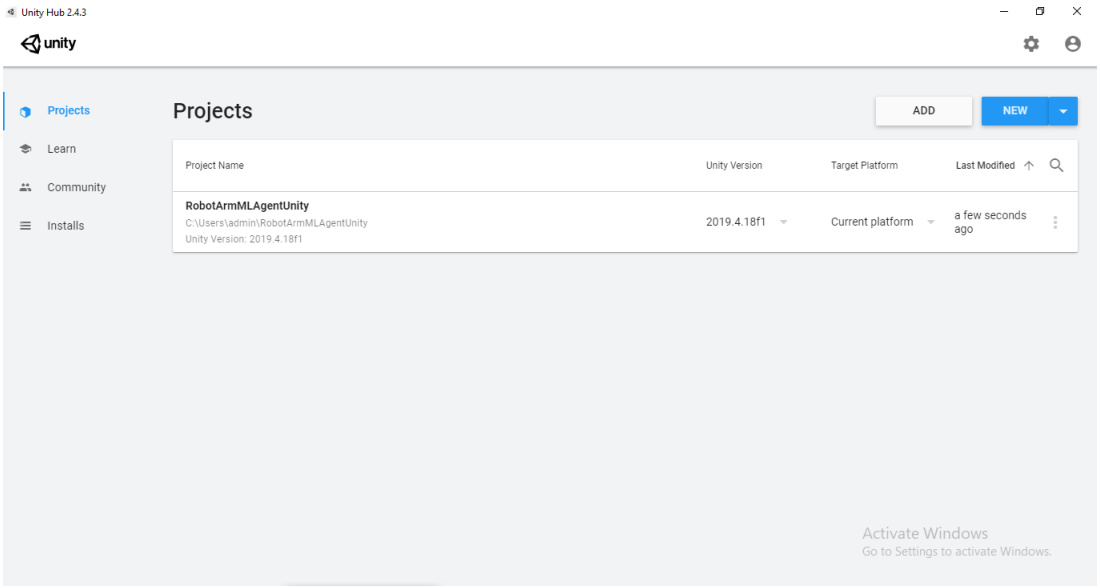


Fig. 1. Unity Hub GUI start-up page.

Ongoing AI innovations have elevated a necessity of building a complex simulation environment analogous to complex physical world. Unfortunately, most simulators lack a complex environment to map to the actual physical world or lack interaction with agents and the environment [27]. Additionally, some simulated environments even have limitations in physics. Thus, the simulators are unable to offer complex challenges to the agents learning in an environment. Latest game development platforms have powerful visual features to handle complex physics and sophisticated interactions. Also, the gaming environments are interactive, simpler to the user, and supportive across other platforms.

The game development platforms like Unreal engine can even serve as a universal platform during the research, but the missing components are the abstraction and interfaces necessary for the research [27].

Unity ML-Agents is the latest development of an open-source package to train the agents in a simulation environment. The toolkit supports training using Python-API and multiple training approaches such as reinforcement learning, imitation learning, and neuroevolution. The package additionally enables TensorFlow-based state-of-the-art for many applications. ML-Agents is not only applicable to AI researchers but also to game programmers for 2D, and 3D AR/VR games. Implementation of trained agent is effortless in real-time. Advances in AI research can be combined to extend the game development community. Therefore, we use the Unity platform as our simulation platform.

When we first launch Unity software we see another GUI page. Fig. 2 shows the Unity scene when we first open the project through Unity Hub (Fig. 1). It has a $GameObject$ that is our $Agent$ or robot. We modified and customized the robot simulation environment for this research. All the interfaces necessary for the DRL application are found in the $ML-Agents$ package that are easily downloadable from package manager in Unity. Behaviour parameters on the right-hand panel are the DRL parameters required for training the agent.

Most of the machine learning frameworks are in Python scripting language. In Unity ML-Agents, the package uses Python and TensorFlow but with some abstraction for the Unity developers. The main advantage here is that one can train the model without actually writing a Python code. Thus, it is the best technique for a novice programmer.

### 2.3 Resources for 6 DoF Machine Learning and Simulation

There are many software resources by Unity that concentrates on pick and place application. However, the main advantage of Unity is the combination with many other workflows for various applications. The recent advances and research are below.

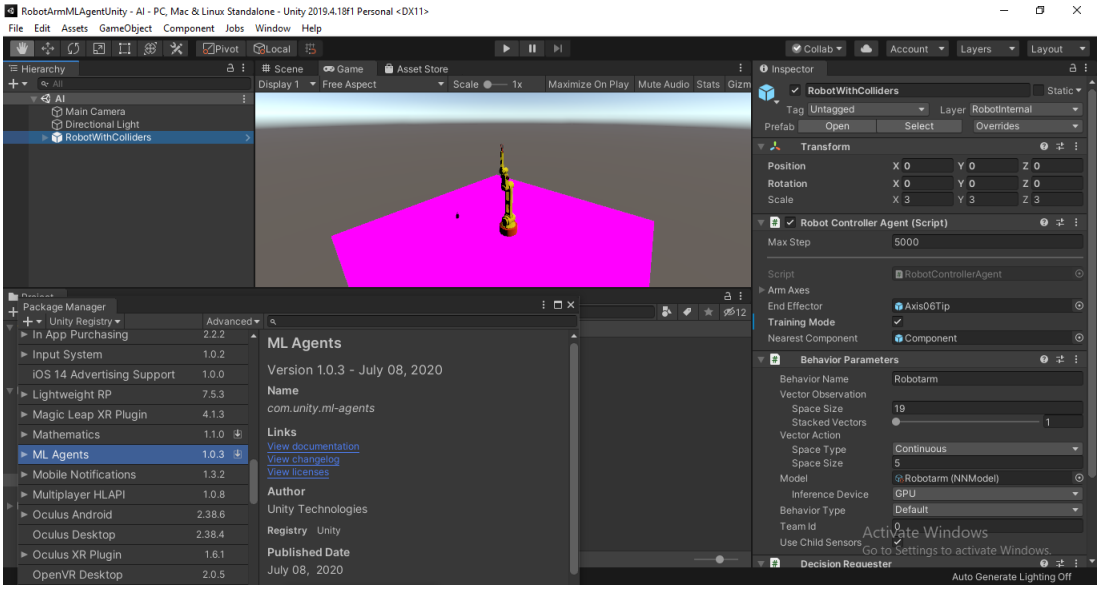


Fig. 2. Unity 3D scene GUI.

1) Pick and Place Tutorial

Unity Robotics Hub describes an integration of Unity and ROS for pick and place application [28], [29]. The author uses the Niryo 6-axis robot and URDF files for the operation [30], [31]. The tutorial is based on sending and receiving messages from ROS and Unity with MoveIt trajectory but lacks DRL [32]. To test out the environment, we have run a test run this platform as shown in Fig. 3.

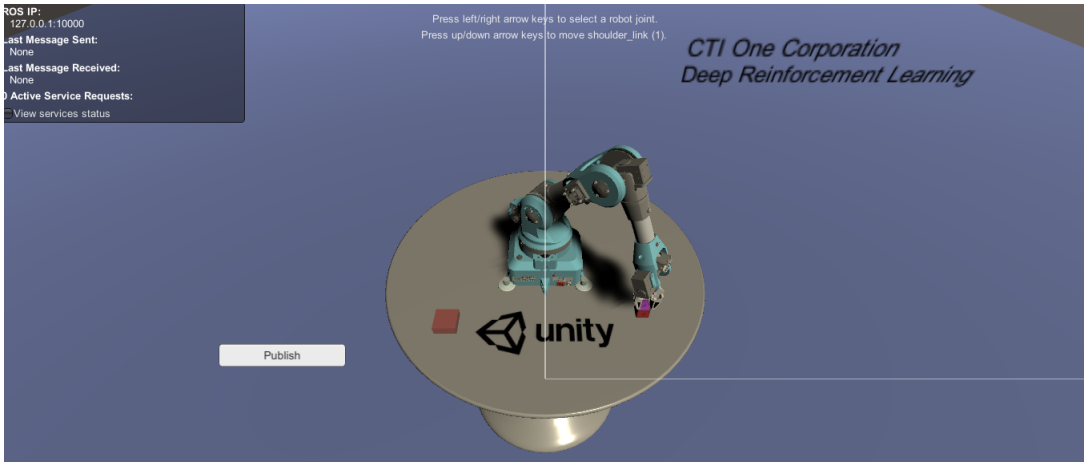


Fig. 3. Unity-ROS simulation without DRL.

2) CV Training using Unity

An official blog by Unity describes how the Unity platform can come in handy when dealing with large-size unlabeled data for CV applications [33], [34]. Usage of synthetic data is one of the solutions for the challenges arising from collecting high-quality labeled data. Unity Technologies overcome this challenge by developing and illustrating high-quality, real-world images.

# 3 METHODOLOGY

## 3.1 DRL Technique for Human-like Robotics Operations

This section refers to a review of a paper on reinforcement learning for robotics applications [35]. The review includes techniques like Trust Region Policy Optimization, DQN with Normalized Advantage Functions, DDPG, and Vanilla Policy Gradient. The techniques implemented in the paper are similar to our research paper, like reaching the target object, picking it, and placing it in a random position. We carry out simulations to verify the paper's findings.

We want to gain good insights into the procedures to precisely estimate the algorithm's hyper-parameters. Therefore, we conduct the experiments on the Unity 3D platform for tuning the hyper-parameters [36]. We will carry out the experiments in our lab environment on 6 DoF robot to demonstrate our designed reward function and a transition to a real-world environment (FD100) for fast, readily deployable results [36].

DRL contains two important blocks, namely an agent and an environment. Fig. 4 shows the block diagram of DRL of our robotics application. It contains two blocks, the agent (robot) and the environment. We can note that the system is a closed-loop system due to feedback. Normally, we have a sensor feedback in a closed-loop system, but there is an additional feedback (reward) in our system, which is one of the most important parameters of any DRL application. The agent collects observations and feeds them to a DNN model, where it learns a policy. The learned policy maps the states to the actions, and thus, we have an output action. In our application, the robot arm moves from one location to another in world coordinate system. The movement is shown in the environment block using two dots in a 3D environment. The coordinate $P_i$ is the current location/state at time $i$ and after taking a certain action, the location is changed to $P_{i+n}$. The agent/robot then receives two types of feedback: a sensor feedback and a reward

feedback. The discussion about sensor feedback is out of the scope of this research. We are focused on the reward for its maximization and accelerated performance.

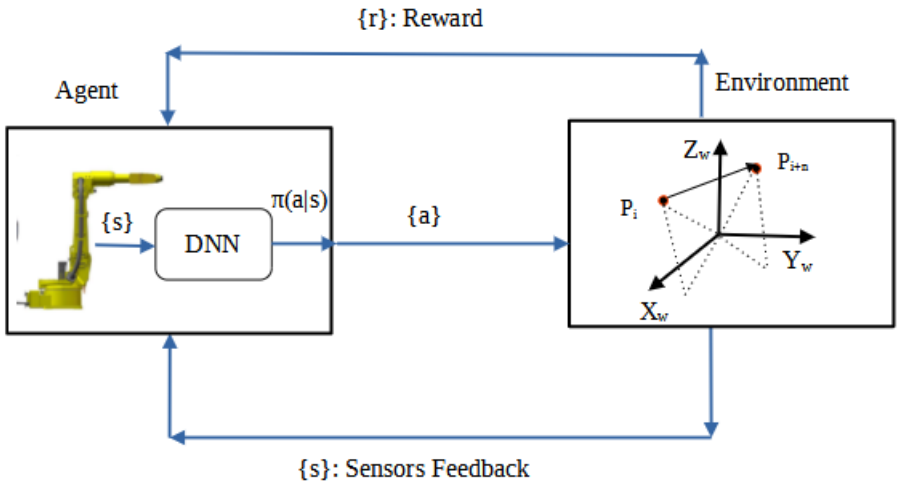


Fig. 4. DRL architecture.

There is a sub-block called DNN in an agent's block. This block becomes a backbone in learning desired tasks. Similar to sensor feedback discussion, this research does not focus on the development of neural network architecture. We use previous researcher's work in our research to build upon it. Fig. 5 shows CNN architecture developed by Mnih et al. [5]. It has three convolutional layers followed by two fully connected neural network layers. Each hidden layer consists of a rectifier activation function. We use their architecture for training our agent.

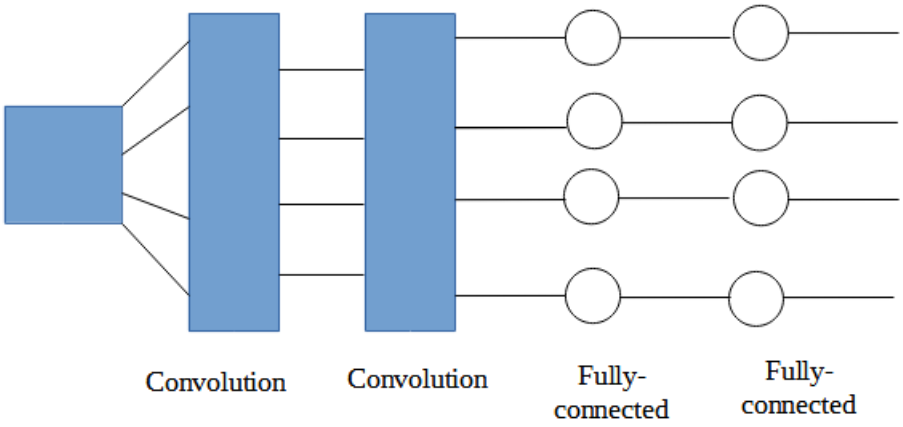


Fig. 5. DNN architecture for human-level control.

### *3.1.1 Mathematical Formulation*

This section provides mathematical formulation for DRL [36]. Robotics task planning consists of sequential moving and controlling of joints and arms of a robot using controllers and motors. The controller controls the movement by defining the values of torque of the motors. Robotics operations are the movement of the parts of the robot in a controlled stochastic environment. The control actions are torques from the controllers. From DRL point of view, the goal of robot operations is to maximize a long-term reward. We modify the existing value function proposed by the Google team and implemented it on Unity. Before jumping to the formulation details, let us first define what a reward function is.

A sequential decision-making process is modeled based on MDP as follows [14]:

1) The robot/agent observes current state,
2) The robot/agent then takes an action according to the observed state,
3) The robot/agent moves to another state by taking action, and

4) Eventually receives a feedback, e.g: reward based on its state to state transition under the action at a time.

Mathematically, the process is formulated as:

1) a set of states S,

   $S = \{s_1, s_2, ..., s_N\}$,

2) a set of actions A,

   $A = \{a_1, a_2, ..., a_M\}$,

3) a transition function,

   $T = S \times A \rightarrow S$, and

4) a reward function

   $R(s_t, a_t) = \{r_{11}, r_{12}, .., r_{MN}\}$

They are denoted as a tuple $< S, A, T, R >$. When in any state $s \in S$, an action will lead to a new state with a transition probability $P_T(s, a, s\prime)$, and a scalar reward $R(s, a)$ function. The stochastic policy $\pi : S \rightarrow A$ maps from state space to a set of actions, and $\pi(a|s)$ represents the probability of choosing an action $a$ at a state $s$. The goal is to find an optimal policy $\pi^*$ to maximize the scalar rewards.

$$\arg\max_{\pi \in \Omega_\pi} \{\text{Exp}[\sum_{k=0}^{H-1} \gamma^k R(s_k, a_k)]\} \tag{1}$$

Final version of a maximized reward function is in the Equation 1. Each term in the Equation 1 is dissolvable. Firstly as a reward function at any time $k$:

$$R(s_k, a_k) \tag{2}$$

The reward function is then discounted by a discount factor $\gamma$ at time $k$. Here $\gamma < 1$ because as time $k$ increases, the rewards tend to become smaller.

$$\gamma^k R(s_k, a_k) \tag{3}$$

Note that in a finite horizon or goal-oriented domains, choose discount factors close to 1 to encourage actions towards the goal, whereas in infinite horizon domains choose a lower discount factor to achieve a balance between short-term and long-term rewards. For all the experiments up to horizon $H$, we get a summation as:

$$\sum_{k=0}^{H-1} \gamma^k R(s_k, a_k) \tag{4}$$

For a stochastic nature of the rewards, we use statistical expectation as:

$$\text{Exp}\{\sum_{k=0}^{H-1} \gamma^k R(s_k, a_k)\} = \int_{inf} \gamma^k R(s_k, a_k) P(s_k, a_k) ds \tag{5}$$

Hence, we have an average discounted rewards under a policy $\pi$.

### *3.1.2 Design Development Platform*

Previous chapter explains that Unity 3D engine is a universal platform for learning agents. Furthermore, Unity provides flexibility in interaction to user and even for agent and environment. Due to high-end physics features, one of the most critical aspects of game development, Unity stands out in DRL applications. User can set up an interactive environment for the agent to learn during simulation process. Despite this, simulation environments often are not as accurate as real world. Nevertheless, Unity has a flexibility to provide a hard time to the agent during the learning process. The agent can have better performance when facing real-world applications. With these insights, Unity team recently developed a novel open-source package known as ML-Agents to make reinforcement learning straightforward [27]. The team claims Unity to be advantageous for the game programmer's community and AI researchers.

This project requires Unity 3D software and Anaconda Python 3 virtual environment with $ml-agents$ package installation [37]. This package is Python 3 package. In Unity, we require another $ML-Agents$ package for an interaction between Unity and Python.

We use ML-Agents version 1.0.3 which is based on TensorFlow API. The latest versions are even compatible with PyTorch API. Unity also supports accelerated training of the agent on a GPU. We used CPU training for our agent because we did not have GPU access.

There are two Unity scenes in our project. AI scene contains only one agent, and training scene contains a farm of agents. We train using the AI scene in our project, i.e., the scene containing only one agent. Fig. 2 shows the AI scene with only one agent. Fig. 6 shows a scene containing twelve agents. Here, each robot is an independent agent, but they all share same behavior. Thus, using multiple agents for training is good but very time-consuming because all the agents collectively do not accelerate the training but contribute to training in parallel. Still, we can easily switch to multiple agents or single agent training using the lowermost panel under *Assets* > *Scenes* in Fig. 6.

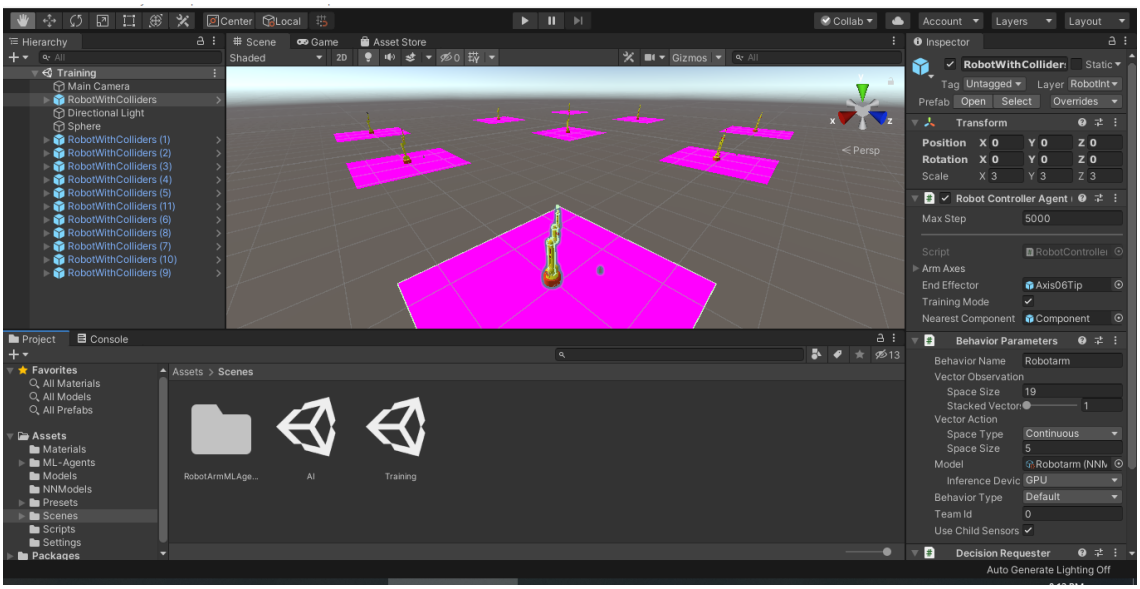


Fig. 6. A farm of 12 robots/agents.

When we first open training scene, the robot's physical color and background are identical, which makes it challenging to differentiate background from the agent [11].

Therefore, we modify shader materials for ease in observation and differentiation between the environment and the agent. This can be changed from Assets in the Project Window. If necessary, we can even change the target object's shader material too.

After saving these changes, we open Anaconda Navigator to start training process. First, the training requires a configuration file. The configuration contains hyper-parameters for DRL such as discount factor, episode length, buffer size, reward signal strengths, and many more. Next, we configure training parameters necessary for the training. Even if we miss any parameter, Unity has a default value for each trainer type which becomes easier for the programmer. It even has settings for neural network architecture, namely number of layers, number of hidden units, batch size, and other parameters. The most important parameter for our training is the algorithm or trainer type-setting. Unity supports two types of trainers: PPO and SAC. The difference between these two trainers is in upcoming sections.

As mentioned earlier, we require a configuration file for initiating the training and setting up all the essential parameters. Fig. 7 shows the training configuration file for our experiments and research. The configuration file has parameters like *trainer_type*, *network_settings*, etc. We customized the configuration file by trial and error method and referring to documentation by Unity team. We say trial and error because sometimes these parameters are important and would work well with the application, especially in neural network applications. Though, we can realize it after actual implementation of the training. Some important settings for our application are below.

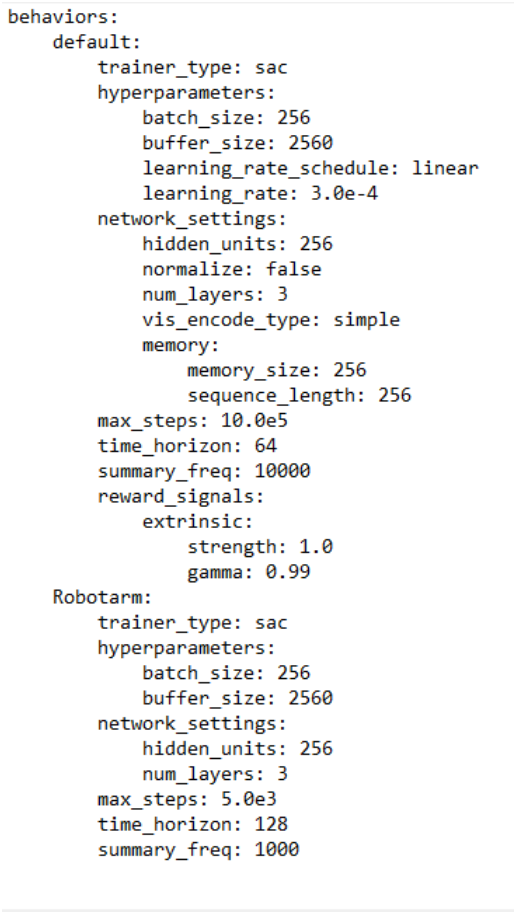


Fig. 7. Training configuration file.

A parameter called *vis_encode_type* is a setting that belongs to a type of network. The network here means neural network for training DRL application. Unity has options to opt for architecture from available options. The options are *simple*, *nature_cnn*, *resnet*, etc. We choose the default settings known as *simple*. It has a simple encoder consisting of 2 CNN layers, an architecture developed by Mnih et al. [5]. Upcoming sections explain this architecture in detail.

Another important network settings parameters are *num_layers* and *hidden_units*. The *num_layers* is a short-hand notation for several layers in a neural network. It adds a given number of hidden layers after an input layer. For our research, we choose three

hidden layers after the input layer. After deciding the number of layers, we choose number of hidden neurons in each layer. We chose 256 units in each layer. We came up with these values after trying simpler networks, such as a fewer hidden units and a fewer number of layers. Unity documentation suggests having a complex network settings if complexity of the task increases. Raju K. suggests doing some trial and error methods before fixing YAML configuration file [11].

Our robot got stuck in certain position sometimes for a longer time and could not come out of it. Fig. 8 shows cobra position of the robot. We encountered this issue and could not figure out how to overcome it before reading an article [11]. This figure is taken from the article [11]. According to the article, the robot arm might get stuck in the cobra position as shown in Fig. 8 if the network settings are simple. The author used a different *trainer_type* (PPO algorithm) for his training. His article suggests increasing *hidden_units* or *batch_size* to make the neural network architecture more complex. Doing so might eradicate the issue of the cobra pose, and the robotic arm might not get lost in the loop.

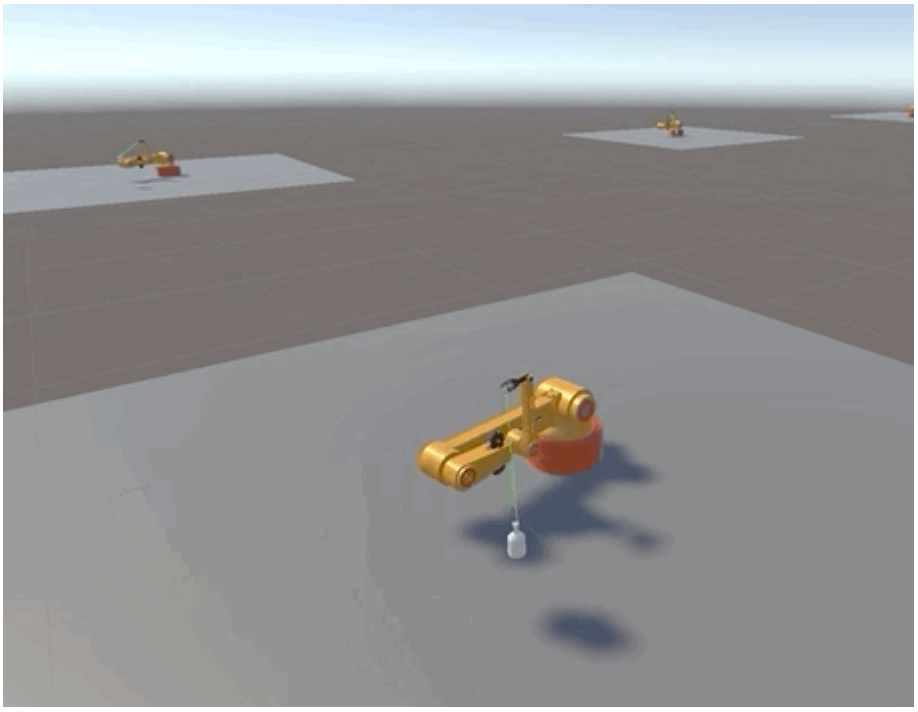

Fig. 8. Cobra position.

*Buffer_ini_steps* is also a vital hyper-parameter. Unfortunately, we do not have a customized hyper-parameter value for our training; therefore, the agent will assume a default value, 0. This hyper-parameter updates policy by collecting several experiences into a buffer. Values higher than 0 resulted in a random behavior of the agent. Regrettably, the agent possesses zero experience during the start of the training, and so the buffer was filled arbitrarily, which raised an issue during comparison. Therefore, we use a default value to make an easy comparison and remove the randomness.

Training any DNN takes millions of steps. Neverthless, due to time limitations, we train it for a few thousand steps. Anaconda console prints out the training status. A typical training looks like as shown in Fig. 9. It starts by printing training configuration parameters to verify them before the start of the training. The console prints out agent's name, number of steps reached, time required to complete those steps in seconds, mean

reward reached for a certain number of steps, standard deviation of the reward, and training status. If we study the mean reward at each step, we see that the mean reward increases as the steps increase. If any of the parameters or hyper-parameters are wrong optimally, there is a deviation from the increasing trend. Sometimes, due to unknown run-time errors during the training, there is an interrupt. However, Unity has a check-point saving feature after every specific interval of time or steps. We can set a value of the *check_point* interval in the training configuration file. Consequently, one can resume the training quickly. After completion of the training, a trained model is saved on the local computer memory. The trained model is saved with a file extension of *.onnx*.

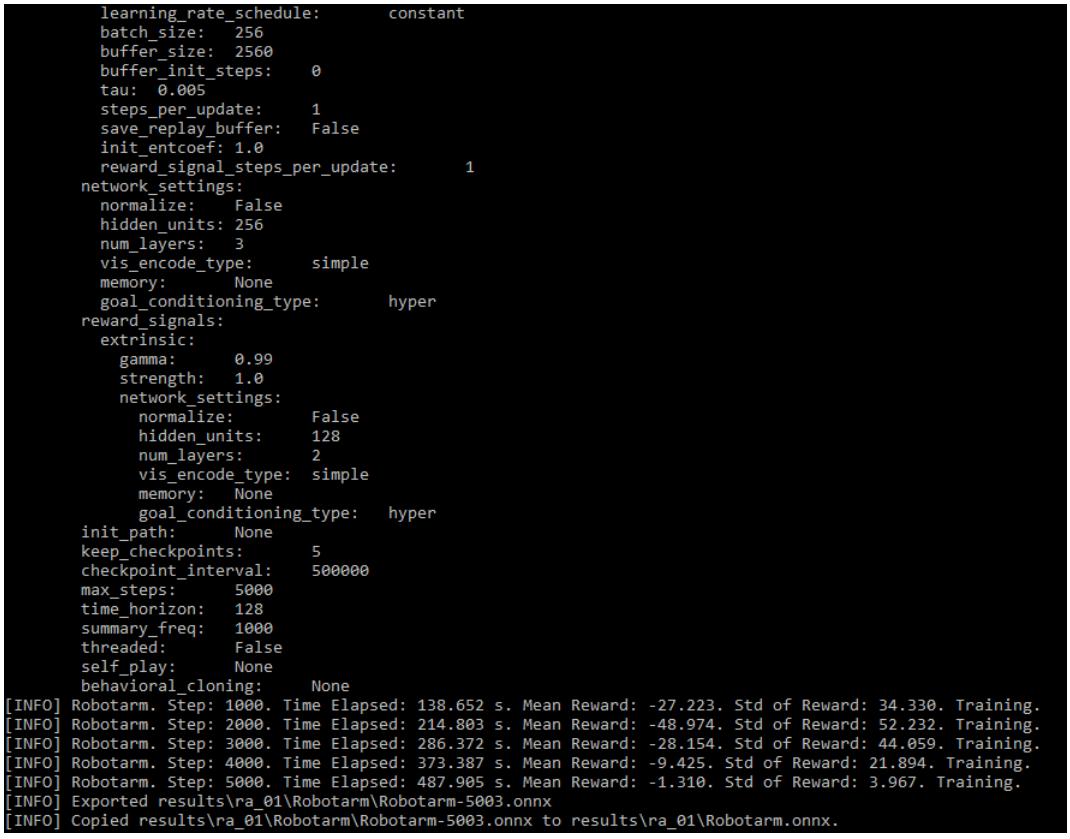


Fig. 9. Training status.

Unity also supports Tensorboard visualization for trained models. The trained model collects and saves training data in a log file which can be retrieved and plotted on Tensorboard. Fig. 10 shows Tensorboard GUI. A left-hand panel in Fig. 10 shows various settings to study and customization of plots. It even shows a legend for the plots. The right-hand panel is for the plots and training status.

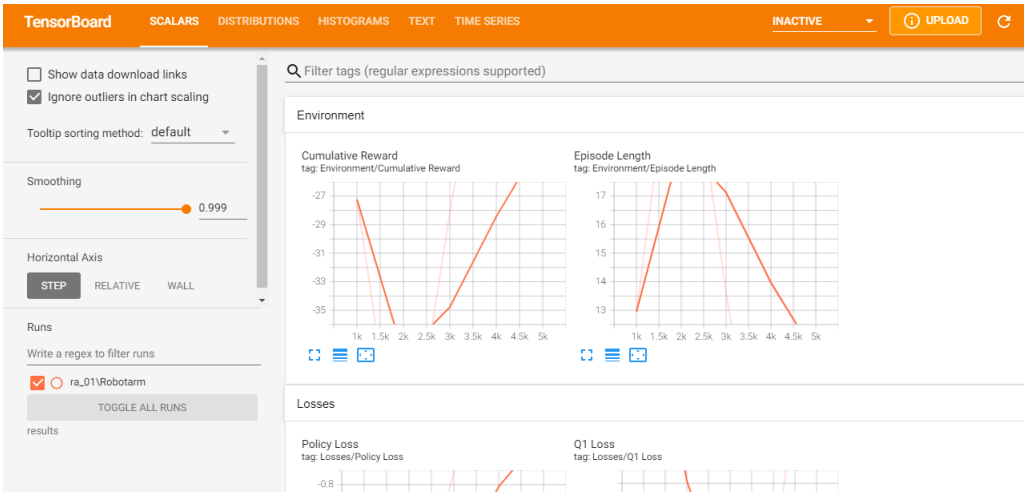


Fig. 10. Tensorboard GUI.

## 3.2 Reward Function and Policy for Robotics Operations

### *3.2.1 Mathematical Formulation for Reward Functions*

The section explains formulation of baseline reward function. The reward function $R(s_k, a_k)$ is directly proportional to the $\Delta$ *distance* [38].

$$R(s_k, a_k) \propto \Delta distance \tag{6}$$

$$\Delta distance(k) = distance(k) - prevBest(k-1) \tag{7}$$

Here, $\Delta$ *distance* is the difference between a current *distance* and *prevBest* as shown in Equation 7.

Now, let us see what *distance* and *prevBest* are. These two are similar as both of them refer to the distance between end effector and target. The *distance* can be calculated as:

$$distance = ||P_{end} - P_{tgt}|| \leq \varepsilon \tag{8}$$

Action and state spaces for our robotic operations are continuous thus the value of *distance* automatically becomes infinite. The equation below states that the range of reward, varies from $-\infty$ to $+\infty$.

$$-\infty < R(s_k, a_k) < +\infty \tag{9}$$

To further elaborate reward and penalty dependency on the $\Delta$ *distance*, we re-write the function explicitly. Here, $P(s_k, a_k)$ is a penalty function that depends on the distance calculated by two values: *prevBest* and *distance*. A *beginDistance* is a distance between the end effector and the target when a new episode begins. It remains constant throughout a particular episode. The *distance* is the distance calculated at each step when the robot arm takes an action. The *prevBest* is the closest distance the robot previously reached. The *prevBest* plays a crucial role in deciding the reward or penalty for our application. The mathematical form and update of *prevBest* are:

$$P(s_k, a_k) \propto -\Delta distance \tag{10}$$

$$R(s_k, a_k) \propto \Delta distance \tag{11}$$

Thus, the baseline reward function is as follows:

$$R(s_k, a_k) = \begin{cases} -\Delta d(k), & \text{if } \Delta d(k) < 0, \\ beginDistance(k) - prevBest(k) - \Delta d(k), & \text{otherwise.} \end{cases} \tag{12}$$

$$prevBest(k) = \begin{cases} beginDistance(k), & \text{if } k = 0 \\ min(distace(k), prevBest(k-1)) & \text{if } k = 1, 2, ..., t \end{cases} \tag{13}$$

Value of *prevBest* is updated according to Equation 13. Further, we introduce gains $k_1$ and $k_2$ to scale up and down the values of reward and penalty for accelerating the reward function. The $k_1$ gain is for the penalty function and $k_2$ gain is for reward function. Note that the values of $k_1$ and $k_2$ are non-negative integers. The equation then becomes as follows:

$$R(s_k, a_k) = \begin{cases} -k_1 \Delta d(k), & \text{if } \Delta d(k) < 0, \\ k_2(beginDistance(k) - prevBest(k) - \Delta d(k)), & \text{otherwise.} \end{cases} \tag{14}$$

### *3.2.2 Design and Development*

The reward function is designed based on a policy $\pi$, which has to satisfy three guidelines: feasibility, intuitive, merit-based. The baseline algorithm has a few loopholes that makes it lesser feasible. Coordinate axes for the reward function is different for both reward and penalty. The penalty is a reward but with a negative value. The algorithm has two linear graphs for reward and penalty. The function should follow a joint base for better realization. Thus, we modify the algorithm so that the reward and the penalty have a common base/axis but are in different quadrants. Also, to maximize cumulative long-term reward, we should make reward as high as possible and vice versa for the penalty. The reward and penalty can be scaled up or down according to different

applications. We propose to keep the reward more prominent than the penalty. Therefore, we introduce $k_1$ and $k_2$ scaling factors in the reward function. These factors define slopes for reward and penalty, respectively. There is a need to standardize the definitions of variables such as *prevBest*, *beginDistance*, *distance*, making the reward function intuitive. The modifications in the reward function that we propose saves values corresponding to its time index. It makes a programmer or a researcher trace-back progress and modifications. According to MDP, time in a DRL application is most important as the agent learns from its past experiences. Therefore, we add a time index buffer for each of the variables. While the baseline model lacks the time index tracking. For example, when the console printed the training progress with reward values and steps, we could not optimize or improve over because we did not know what was going on in the back of the hood. We modified the code to export the data to a CSV file. The file contained all the values of important variables at each step/time. It even tracked the start and end of the episode, including when the agent received a hefty penalty and reward. This intuitive nature helped us modify the reward function and take appropriations.

## 3.3 Design of Type-I and Type-II Reward Functions

### *3.3.1 Type-I Baseline Reward Function*

As discussed earlier, the baseline reward function follows two different graphs. The reward function, i.e., a positive reward, is dependent on a difference between *beginDistance* and *distance*. In comparison, the penalty is a difference between *prevBest* and *distance*. We changed the baseline reward function to make a common coordinate, and now the baseline looks like in Fig. 11. This reward function is easy to understand and compare. The graph follows Equation 12 and implies that as the $\Delta distance$ increases, the negative reward or the penalty increases. Similarly, as the $\Delta distance$ decreases, the positive reward increases.

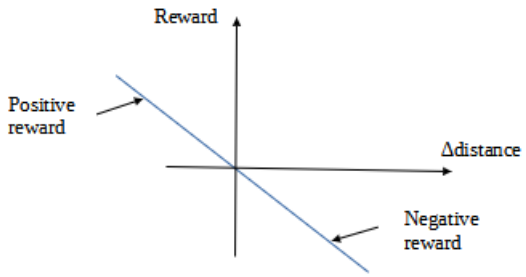


Fig. 11. Baseline reward and penalty functions.

### *3.3.2 Type-II Modified Reward Function*

This section explains development of modified reward function. The modified reward function graph is shown in Fig. 12. The graph follows Equation 14 and has three different values of slope. As value of $k_1$ slope increases, received penalty also increases. Here, we are more focused on the maximization of the positive reward than the negative reward. Therefore, we keep the penalty gain $k_1$ constant as our baseline model. The values of $k_2$ are changed to achieve the highest possible reward. With $k_2 = 1$, our modified function behaves like original model. We experiment with the behavior of the algorithm and agent by changing or more explicitly increasing the $k_1$ value. .

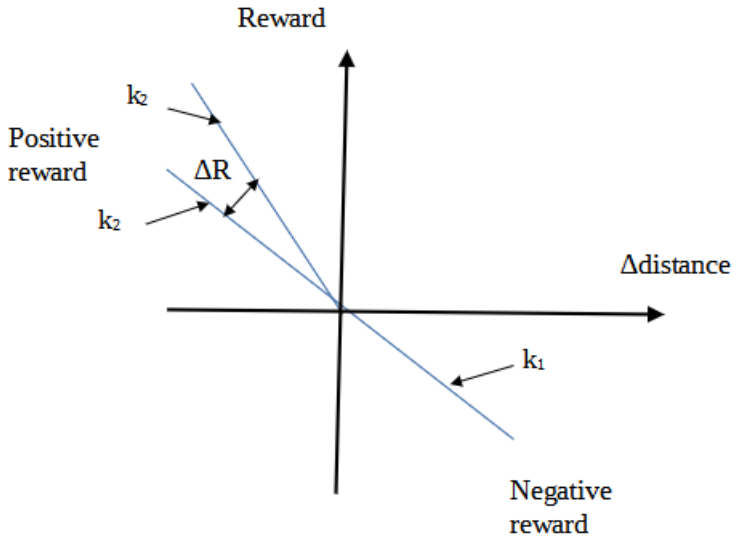


Fig. 12. Illustration of modified reward function.

### 3.4 SAC Technique

We discussed MDP in previous sections. MDP has a short-term memory that allows us to get the future states with a help of history. Policy guides an agent to the future states by recalling this history [38]. Now, we discuss policy and SAC algorithm by Google DeepMind in this section. A policy is a guideline that rewards the agent based on its performance. SAC follows a stochastic policy gradient that operates in a stochastic environment. The stochastic environment here means a randomized environment. For example, considering our application, the robot is expected to reach a particular position by taking an action. However, due to some unexpected random behavior, the robot fails to achieve this goal. This behavior makes it stochastic. The stochastic policy works better with the continuous environment. The Fig. 13 shows an overall scope of the stochastic policy and a relationship to other algorithms.

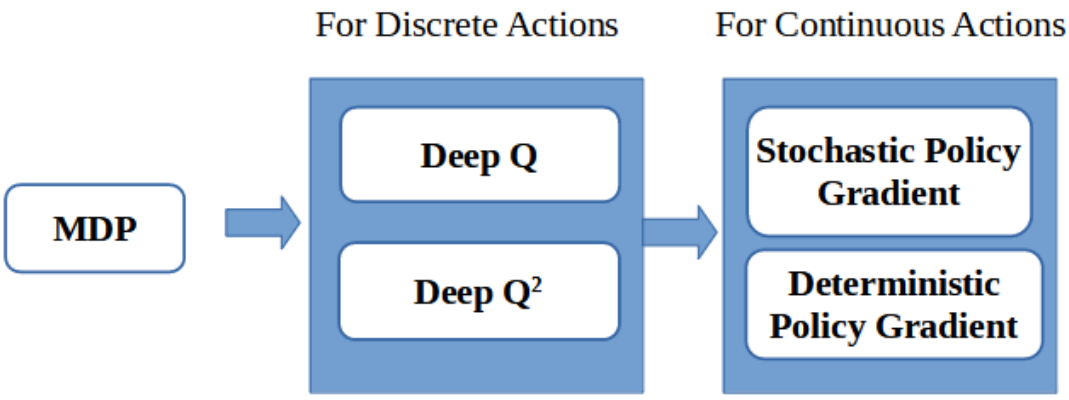


Fig. 13. Scope of stochastic policy algorithm.

The MDP has a "short-term" memory for any state $s_t$.

$$Prob(s_{t+1}|s_t) = Prob(s_{t+1}|s_t, s_{t-1}, ..., s_2, s_1). \tag{15}$$

Reward function is as follows:

$$J(\theta) = \sum_{s \sim S} Prob(s) \sum_{a \sim A} \pi_\theta(s,a) R(s,a). \tag{16}$$

We substitute a policy $\pi$ with a conditional probability in Equation 16. Thus, the equation finally changes to:

$$J(\theta) = \sum_{s \sim S} Prob(s) \sum_{a \sim A} Prob(a|s,\theta) R(s,a). \tag{17}$$

This conditional probability in Equation 17 connects to the MDP. To maximize the reward, we use stochastic policy gradient. So, we take gradient on both sides. Moreover,

the equation we get is:

$$\bigtriangledown J(\theta) = \sum_{s \sim S} Prob(s) \sum_{a \sim A} Prob(a|s,\theta) \bigtriangledown log(Prob(a|s,\theta)) R(s,a). \tag{18}$$

### 3.5 SAC vs PPO

Unity ML-Agents package previously supported only one DRL training algorithm known as PPO [25]. The PPO algorithm developed by OpenAI team has benefits in terms of complexity and tuning of hyper-parameters [39]. Nevertheless, a recent development of SAC algorithm shows outstanding results when compared to other continuous action space algorithms [9]. Therefore, to accelerate training of agents, Unity released a new version of ML-Agents package with a support of the SAC algorithm. Besides, SAC proves to be sample efficient, which means that we need less time to learn the policy.

There are two types of algorithms for DRL, namely an on-policy and off-policy algorithms. PPO uses on-policy technique in which the model learns and improves the policy through collecting samples. Eventually, it increases the chances of rewarding actions than the actions that are non-rewarding. The on-policy algorithms estimate a reward using an evaluation function. In addition, these evaluation functions estimate the accumulative reward if that particular policy is applied.

There is a different case for off-policy algorithms like SAC. Instead of learning the current policy, the evaluators learn across all the policies. The SAC uses all the previously learned samples more efficiently since the start of the time, making off-policy algorithms more sample efficient. The usage of these old samples is called experience replay stored in a buffer known as replay buffers.

# 4 ALGORITHM IMPLEMENTATION

In previous sections we presented methodology of our research. Now, we discuss an actual implementation of reward function in terms of an algorithm.

## 4.1 DRL Implementation

Unity setup for a 6 DoF robot is shown in Fig. 14. It shows different axes of the robot, a target object, and a ground plane where both the objects are placed [11]. This figure representation is from an article [11]. The motors and gears drive each axis in the robot's configuration. Movements of each joint are in a range of 0° to 360°. Here, words like axis and DoF used in this paper are interchangeable.

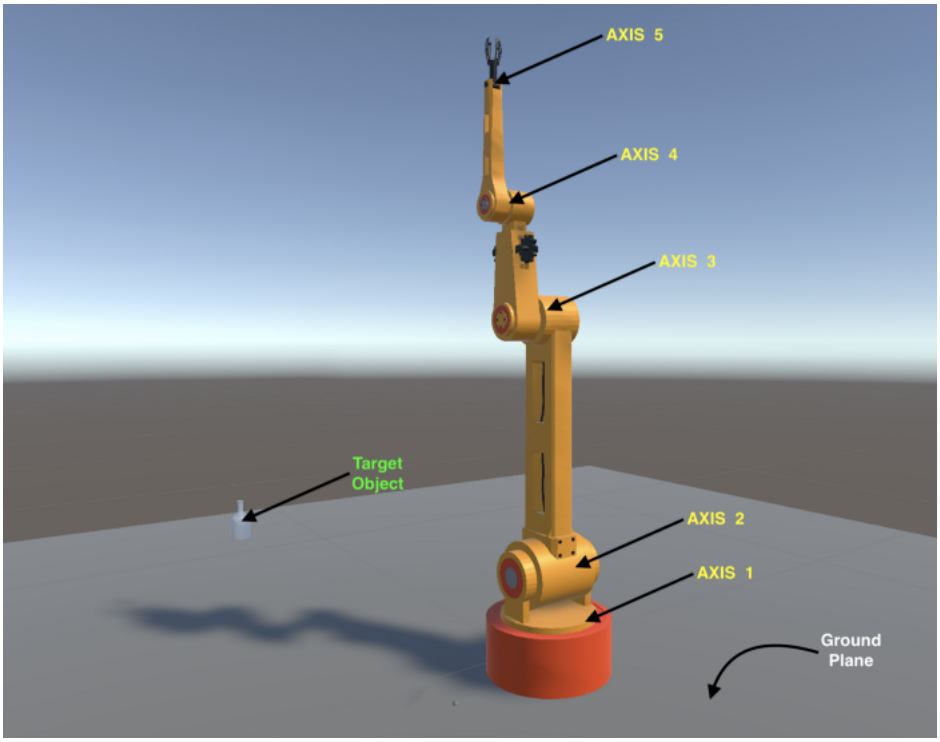


Fig. 14. Unity setup of 6 DoF.

Unity ML-Agent training framework supports MDP. It works in following ways:

1) Step 1: The agent observes/sees an existing state. Vision cameras and robot sensors generally record these observations.

2) Step 2: The agent takes an action. The action can be a movement of a joint or a combination of joints.
3) Step 3: The agent gets a feedback. The feedback is in a form of a scalar reward or penalty depending upon the action taken.
4) Step 4: Process repeats from Step 1 until the end of an episode.

We map all the above steps to ML-programming. Following Table 1 shows the mapping. States in an MDP corresponds to a collection of observations. It implies to a current state of the robot. In Unity, a typical program calls *OnActionReceived* function to complete a task of taking some action. The robot moves from the current state to the next state. Consecutively, the robot receives a feedback for that taken action. The feedback can be a positive scalar reward or a negative scalar reward. *AddReward* function adds the reward and accumulates until the end of the episode.

TABLE 1

Programming Aspects for ML

| Category | Description | Note |
|---|---|---|
| State | CollectObservations() | Agent studies the environment |
| Action | OnActionReceived | Agent takes an action |
| Reward | AddReward | Agent receives reward/penalty |

Every DRL application starts with creating state, action, and reward tables. The state table of 6 DoF robot is shown in Table 2. These are the observations that an agent sees in the environment, such as positions of the target object and gripper. There are a total of 19 states for our application. These states have continuous values.

TABLE 2

State Table for FD100 Robot

| Category | Description | Note % Improvement |
|---|---|---|
| $s_{j1-angle}$ | $[0^\circ, 360^\circ]$ | Continuous, Rotate |
| $s_{j2-angle}$ | $[-90^\circ, +90^\circ]$ | Continuous, Bend |
| $s_{j3-angle}$ | $[-180^\circ, +180^\circ]$ | Continuous, Bend |
| $s_{j4-angle}$ | $[-90^\circ, +90^\circ]$ | Continuous, Bend |
| $s_{j5-angle}$ | $[-90^\circ, +90^\circ]$ | Continuous, Bend |
| $s_{R_x}$ | 0 | Fixed, Robot's base |
| $s_{R_y}$ | 0 | Fixed, Robot's base |
| $s_{R_z}$ | 0 | Fixed, Robot's base |
| $s_{O_x}$ | 0.45 | Fixed, Object's position |
| $s_{O_y}$ | 0.20 | Fixed, Object's position |
| $s_{O_z}$ | 0.45 | Fixed, Object's position |
| $s_{E_x}$ | $(1.1(min\_angle) + 0.9(max\_angle)/2)$ | Fixed, Gripper's position |
| $s_{E_y}$ | $(1.1(min\_angle) + 0.9(max\_angle)/2)$ | Fixed, Gripper's position |
| $s_{E_z}$ | $(1.1(min\_angle) + 0.9(max\_angle)/2)$ | Fixed, Gripper's position |
| $s_{D_x}$ | $(-\infty, +\infty)$ | Continuous, $O_x - E_x$ |
| $s_{D_y}$ | $(-\infty, +\infty)$ | Continuous, $O_y - E_y$ |
| $s_{D_z}$ | $(-\infty, +\infty)$ | Continuous, $O_x - E_x$ |
| $s_D$ | $(-\infty, +\infty)$ | Continuous, Scalar Distance |
| $steps$ | 5000 | Fixed, Step count |

Initially, target position and gripper angles varied in each episode. They were randomized when a new episode started. However, the experimental results seemed to be an arduous task to compare with such stochastic nature. Therefore, we fixed the position of the target in every episode. Similarly, the random assignment of the gripper position and angles at the start of any episode made it hard to differentiate. So, we made the starting point of the gripper constant.

Joint information of the FD100 robot is shown in Fig. 15. As we know, it is a 6 DoF robot. We can verify that there are a total of six joints with different degrees of movement [36]. The figure is from Turin robot user guide [40]. The joint movements are the actions taken by the robot arm when encountering a specific state. The information of each joint movement are:

Axis-1: bottom-most axis: $0^\circ$ to $360^\circ$ rotation

Axis-2: first bend axis: $-90^\circ$ to $+90^\circ$ bend

Axis-3: second bend axis: $-180^\circ$ to $+180^\circ$ bend

Axis-4: third bend axis: $-90^\circ$ to $+90^\circ$ bend

Axis-5: fourth bend axis: $-90^\circ$ to $+90^\circ$ bend

Axis-6: gripper axis: Open and Close

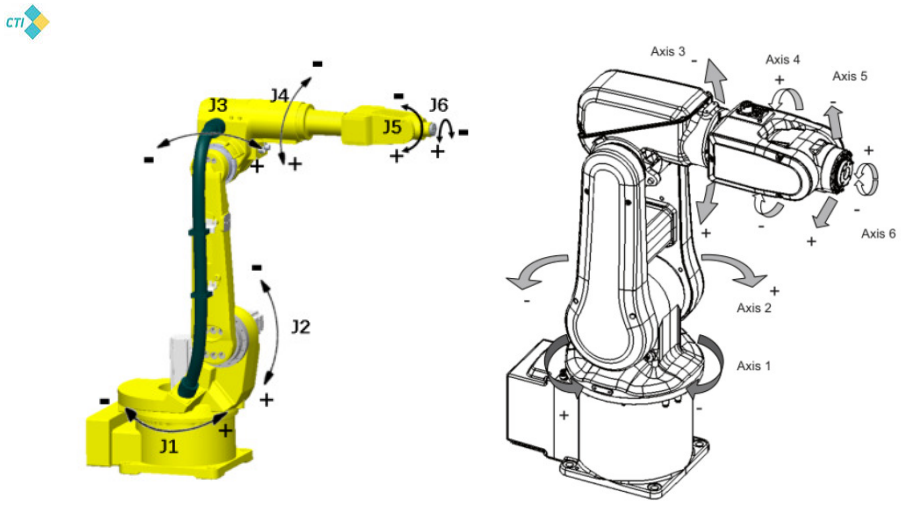


Fig. 15. Turin robot joints diagram.

After building a state table, we build an action table. Here, we have continuous actions for each state. It means that how much a joint should bend or rotate. For example, for discrete car driving games, one can steer right, left, or straight. To make it into

continuous space, we characterize it into a fuzzy membership function that allows us to steer in a right amount as in how much right. We use the latest development, SAC off-policy algorithm for our continuous action space. The action table for a 6 DoF robot is shown in Table 3. The agent moves its joints to take actions and to reach the target object.

TABLE 3

Action Table for FD100 Robot

| Category | Description | Note % Improvement |
|---|---|---|
| $a_{j1-angle}$ | $[0°, 360°]$ | Continuous, Rotate |
| $a_{j2-angle}$ | $[-90°, +90°]$ | Continuous, Bend |
| $a_{j3-angle}$ | $[-180°, +180°]$ | Continuous, Bend |
| $a_{j4-angle}$ | $[-90°, +90°]$ | Continuous, Bend |
| $a_{j5-angle}$ | $[-90°, +90°]$ | Continuous, Bend |
| $a_{gripper}$ | $[Open, Close]$ | Continuous |

For every action the agent takes at each time-stamp, it has to bear some consequences. Accordingly, we create a reward (cost) table to link each state to the corresponding action. A reward can be positive or negative. It is generally normalized in a range between -1 to 1, where -1 is called a hefty penalty and +1 is called a hefty reward. Reward table is shown in Table 4. In our case, when the robotic arm touches the target object, the agent receives a +1, a hefty reward, for successfully reaching the desired goal. Also, when the arm hits the ground, we penalize the agent by giving a -1 reward, which is a hefty penalty.

TABLE 4

Reward Table for FD100 Robot

| Category | Description | Note % Improvement |
|---|---|---|
| *Arm hits the ground* | -1 | *Hefty Penalty, End of episode* |
| *Arm reaches the target* | +1 | *Hefty Reward, End of episode* |
| *Arm moves closer* | *+d* | *Marginal Reward* |
| *Arm moves away* | *-d* | *Marginal Penalty* |

Here, we monitor the gap between the end effector and object to reward or penalize the agent. It is where the baseline reward function and modified reward function differ. A marginal reward in the modified reward function is more than the marginal reward in the baseline reward function. Also, the penalty is more minor in the modified reward function than the baseline reward function. It is so because our primary goals are to maximize long-term reward and to make learning process faster.

When the agent shows a desired behavior, it receives a positive marginal reward and vice versa when it shows an undesired behavior. The desired behavior means the arm tries to move closer to the object as the time-steps increase. If the gap between the gripper and the target increases, we consider it as a non-desired behavior and penalize the agent.

An introduction of rewards makes the agent collect or accumulate more than losing them. So, as training proceeds, the agent learns to earn the rewards more than losing them. Closer the arm moves towards the target object, more positive rewards it accumulates.

After defining states, actions, and corresponding reward function, we create a C# script called *RobotControllerAgent.cs*. We override ML-Agents default Monobehaviour to Agent. First step to override is to ”Initialize,” where we reset all the axes to their default position and rotation. By doing so, we place the gripper position above ground plane. A code snippet in Listing 1 depicts the same. Also, if we are not in training mode, the value

in the MaxStep variable is given as a zero. Function $MoveToSafePre-definedPostion$ will allocate the target position on the ground plane within the reach of the robotic arm. As discussed earlier, the position was random but for our convenience we fixed it. Each episode will start from the same position of the target. The exact position can be known from source-code in Appendices B and C, and Table 2.

Listing 1. Initialization at every episode

```
65    public override void Initialize()
66    {
67        ResetAllAxis();
68        MoveToSafePre-definedPosition();
69        if (!trainingMode) MaxStep = 0;
---
84     }
```

### 4.2 Google Team's Policy on Reward Function

We take reward function from an article based on Google DeepMind's work with Unity 3D simulation as our base algorithm [11]. The original algorithm follows PPO, but we change it to SAC [25]. There are important changes to take care of when switching the algorithm. The main change comes in the training configuration file. A detailed description of the configuration file is in Appendix A. The reason why we switch to the SAC algorithm from the PPO is in upcoming sections.

A pseudocode for the baseline reward function is in Algorithm 1. Listing 2 shows a part of the baseline reward function. We can note that the reward is linearly dependent on the difference in the distance. The reward function here follows a similar pattern as the reward table shown in Table 4. A tested and verified C# script is in Appendix B.

**Algorithm 1** Baseline Algorithm

**Require:** System, MLAgents, UnityEngine

**Ensure:** Reset all Axes, Move to a safe Position, Training Mode;

**while** Episode **do**
  Observe the current *State*
  Take an *Action*
  Update *distance*
  Compute $\Delta distance = distance - prevBest$
  **if** $distance > prevBest$ **then**
    $Reward = -\Delta distance$
  **else**
    Update $\Delta R = beginDistance - prevBest$
    $Reward = (\Delta R - \Delta distance)$
    Update *prevBest*
  **end if**
  Add *StepPenalty*
  Add *Reward*
  **if** Hit the Ground **then**
    End of the Episode
  **end if**
**end while**
Start a new Episode

Listing 2. Baseline Reward Function

```
92    public override void
          OnActionReceived(float[] vectorAction)
93    {
...
111   float diff = beginDistance - distance;
112
```

```
113   if (distance > prevBest)
114      {
115      // Penalty if the arm moves away from target
116        AddReward(penalty);
117      }
118      else
119      {
120      // Reward if the arm moves closer to target
121        AddReward(reward);
122        prevBest = distance;
123      }
124      AddReward(stepPenalty);
125       }
126    }
```

There are a few pre-requisites from Algorithm 1; System, ML-Agents, and Unity Engine packages. After fulfilling these requirements, the code initializes robot's different axes by moving them at a fixed pre-defined starting position. Unity engine even checks different types of modes. A training mode is where we carry out the training process and export the trained model in a results folder under a main project folder. The inference mode is where we can deploy the trained model and study the training status on Unity simulation environment.

*BeginDistance* is a distance between the end effector and the target object when an episode starts. It is constant throughout the episode. For example, we reset the position of a robotic arm when an episode starts. So the value of *beginDistance* also remains the same for all the episodes. *PrevBest* distance at that time-stamp is equal to the *beginDistance* because we expect the robotic arm to move closer and not away from the target. Note that the *beginDistance* remains constant throughout that particular episode, but *prevBest* might change.

At each stage, the robot arm will take action, and value of *distance* will vary accordingly. Thus, if we observe Listing 2, we can assume that the reward depends upon the $\Delta$ *distance*. The robot will receive a negative reward, i.e., penalty when it moves away from the target or distance increases concerning the *prevBest* distance and vice versa happens when it moves closer. When the arm moves closer, the value of *prevBest* updates

for future reference. Due to such reason, we want the robot to not exceed this value. If not, the agent will receive the penalty. The mathematical form is already explained in Equation 12.

### 4.3 Enhancement of Reward Function

In previous section, we explained the baseline reward function algorithm. Now, we about the improvements we make to enhance the reward function. This section discusses modified reward function as an enhanced version of baseline reward function.

If we study the relationship between values of distance and reward, then we can interpret that the reward function has infinite values. This way, we can easily make assumptions and interpretations. To study these values, we tried to normalize the reward function.

$$R(s_k, a_k) \propto \Delta distance \tag{19}$$

$$-1 \leq R(s_k, a_k) \leq +1 \tag{20}$$

The normalized values will only lie between $[-1, 1]$. It is shown in Equation 20. However, this research aims to maximize cumulative long-term reward, and normalizing would not allow the agent to receive more rewards than the baseline reward function. In this scenario, we assumed that the baseline algorithm would perform better than our algorithm.

Then, we square the reward function to accelerate the process.

$$0 \leq R^2(s_k, a_k) \leq +1 \tag{21}$$

A squared reward function looks like Equation 21. We would always get a positive reward if we used the squared reward function and it would be difficult to penalize the robot if there is no negative reward. Thus, we dropped that idea too.

$$-1 \leq R^n(s_k, a_k) \leq +1, \quad where \quad n = 1, 2, .., k \tag{22}$$

Similarly, we observed that the value of $n$, the reward function value, would not exceed $-1$ or 1. Equation 22 describes it mathematically.

These intermediate steps were necessary for studying the nature of the reward function. We got an idea to introduce the gain factors, to accelerate the training process and maximize the reward function. Baseline reward function is approximately a linear function from Fig. 11. The reward function is linearly dependent on $\Delta distance$. We assumed that changing the value of slope by scaling the $\Delta distance$ might help us achieve the goal. Fig. 12 explains modified reward function graphically.

We introduce two gain factors $k_1$ and $k_2$ to form our modified algorithm. The $k_1$ gain factor is a scaling factor for a penalty line. The $k_2$ gain factor is a scaling factor for a reward line. Equation 14 shows mathematical formulation of the modified reward function. The modified reward function behaves similar to the baseline when both the values of $k_1$ and $k_2$ are equal to 1.0.

We experiment with different values of $k_1$ and $k_2$ and finally come up with two values. We fixed the penalty gain-factor as 1.0 and opted for two different values of $k_2$. The two $k_2$ values are 1.1 and 1.5. Our assumption with these values is that the modified reward function with $k_2 = 1.5$ would have the best performance among all. Also, the modified reward function with $k_2 = 1.1$ would perform slightly better than the baseline but worse than the reward function with $k_2 = 1.5$. Thus, under these assumptions and enhancement techniques, we formulate the modified reward function.

## 4.4 Algorithm Development for Reward Function

We discussed mathematical formulation of the modified reward function in the previous section. We also mentioned how we try to enhance the baseline algorithm to get maximized performance. Algorithm 2 shows the modified version of the reward function. Definition of various variables remains the same as explained in the Algorithm 1. Values $k1$ and $k2$ are different, which can be fixed such that the $k1 < k2$. We conduct experiments using distinctive values, keeping this relationship in mind.

Algorithm flow for modified reward function is similar to the baseline algorithm. Difference is only in the formulation of each reward functions. Modified algorithm also starts by checking the pre-requisites for implementing the code, such as System requirements and Unity packages. Finally, it ensures that all the robot's axes are reset with an end-effector position above the ground and the target places in a fixed position.

Training mode continues over a *while* loop until maximum number of steps are crossed. The maximum steps are not the maximum length of an episode. Instead, these are the total number of steps we want to train our robot. The steps are independent of the episodic steps. Therefore, the *while* loop keeps track of the episodic steps, and the training mode counts the overall training steps.

**Algorithm 2** Modified Algorithm with Two Line Segments

**Require:** System, MLAgents, UnityEngine

**Ensure:** Reset all Axes, Move to Safe Position, Training Mode;

**while** Episode **do**
  Observe the current $State$
  Take an $Action$
  Update $distance$
  Compute $\Delta distance = distance - prevBest$
  **if** $\Delta distance < 0$ **then**
    $Reward = -k_1 \cdot \Delta distance$
  **else**
    Update $\Delta R = beginDistance - prevBest$
    $Reward = k_2 \cdot (\Delta R - \Delta distance)$
    Update $prevBest$
  **end if**
  Add $StepPenalty$
  Add $Reward$
  **if** Hit the Ground **then**
    End of the Episode
  **end if**
**end while**
Start a new Episode

Pseudo-code for the modified algorithm developed in Algorithm 2 is shown in Listing 3. The pseudo-code is a part of the modified algorithm script. A detailed source code is in Appendix C.

Listing 3. Modified Reward Function

```
92    public override void
           OnActionReceived(float[] vectorAction)
93    {
...
111   float diff = beginDistance - distance;
112
113   if (distance > prevBest)
114      {
115      // Penalty if the arm moves away from target
116        AddReward(-k1*penalty);
117      }
118      else
119      {
120      // Reward if the arm moves closer to target
121        AddReward(k2*reward);
122        prevBest = distance;
123      }
124      AddReward(stepPenalty);
125       }
126    }
```

We apply MDP here; the robot takes observations from the environment as shown in Table 2. Then, the robot/agent takes an action by moving at least one of the axes as shown in the Table 3. Due to this movement, the value of *distance* changes. We calculate $\Delta distance$ value using the current *distance* and *prevBest*. Equation 7 shows the calculation. If the calculated value of $\Delta distance$ is less than 0 then $if$ statement will be implemented. The $if$ statement corresponds to a penalty. Contrasting to the $if$ statement, if the value of the $\Delta distance$ is greater than zero, then an *else* statement will be implemented. For this scenario, we have a positive reward. We update the *prevBest* value, accumulate the reward, and add a penalty for the step. The loop continues till the end of an episode. The episode ends after the end effector touches the ground or maximum steps are crossed.

### 4.5 System Setup

This section describes a setup information of 6 DoF FD100 industrial robots at Palo Alto, California site. Fig. 16 shows a picture of the robot. There are two main divisions in the setup; first division is Node A in Fig. 17, and second division is Node B in Fig. 18.

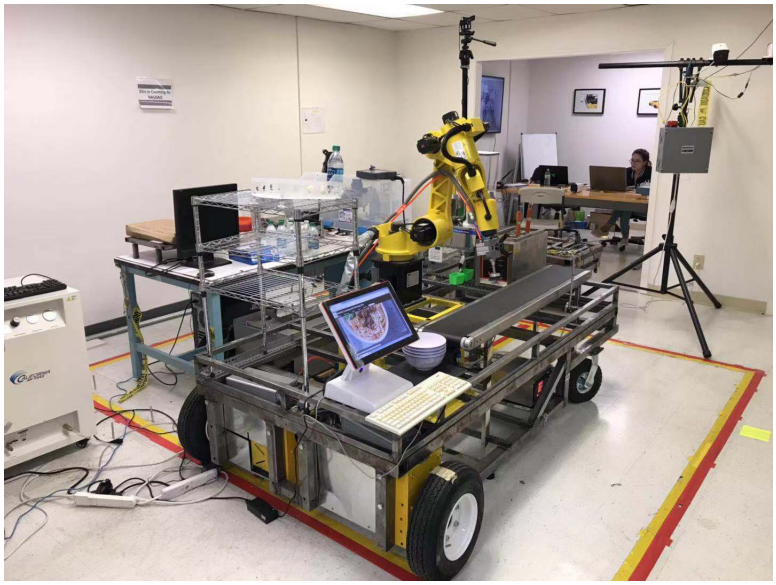

Fig. 16. 6 DoF FD100 robot setup.

Robotic arm of FD100 connects to a controller that processes and controls the movement as shown in Fig. 17. A server (Ubuntu Linux 16.04) connects to the robot using an ethernet connection. The server is then connected to a bridge wirelessly—this configured bridge is a wireless access point between the robot and the server. A principal purpose of this bridge is to extend a LAN using IP addresses within a segment. This MODEM (gateway) directly connects to an IP cloud or MAN.

Node A connects to Node B via MAN in Fig. 18. A wireless connection between Server B and Node A is via a gateway/router access point. Here, we have one more server as ”Supervisor” connected to a router in Node A. That makes two servers now at the Node A; A1 and A2. To establish a connection between A1 and A2, we run a NM program. As the name ”Supervisor” (Server A1) suggests, it supervises or controls the A2 server. The A2 server physically connects to the robot, and the A1 server allows remote access and control of the robot. The Server A1 has all required python scripts to make the robot move.

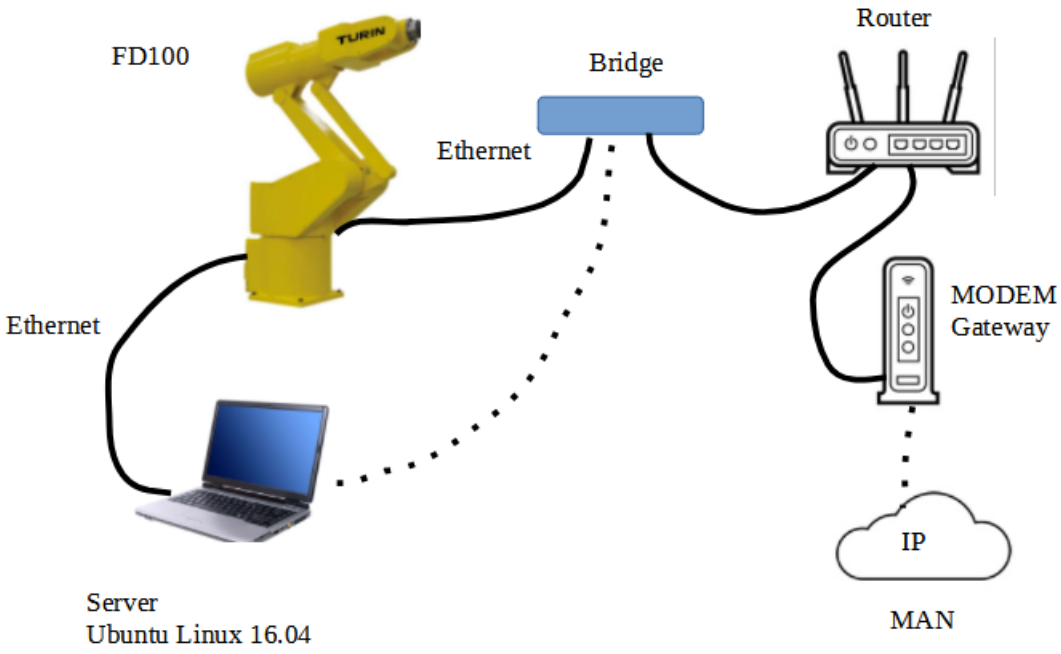


Fig. 17. Robot setup at Node A.

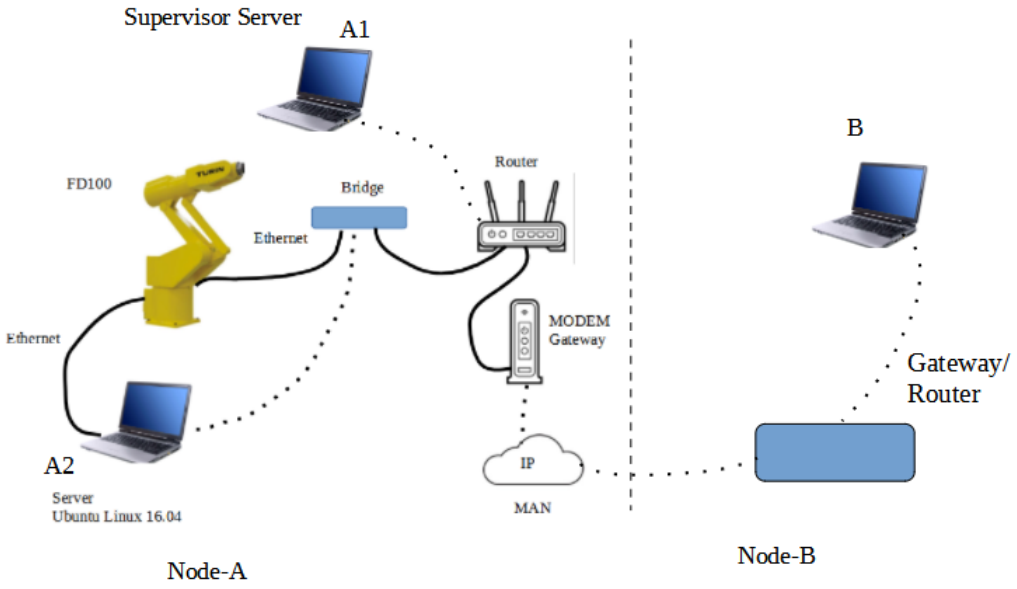


Fig. 18. Node B set up with Node A.

# 5 EXPERIMENTAL RESULTS

## 5.1 Experiment Design Implementation for Reward Function

We perform experiments with baseline and modified reward functions. First, we train our robot on Unity simulation platform twice for each reward function for 5000 steps. We perform the training twice because we have random initialization of weights of neural network, and therefore, the graphs start with a random initial values for each training. Nevertheless, as the training proceeds, the neural network tries to reach optimal weights and each curve tries to overlap or even perform better. The reward values that differ initially are detailed below.

Second, due to less evidence of the performance of reward function, we further decide to train each reward function ten times, to eventually get a total of 20 plots. This time we increased the number of steps and fixed starting and ending positions of target object and end effector gripper. We fixed the positions to remove randomness in our experiments and to evaluate results.

Third, using ten different plots for each reward function, we find an average plot. Eventually, we end up having only two plots from a total of 20 plots. This allows us to calculate the performance of the reward function better.

Last, with these two plots, we can see a visual performance difference. However, for research work, we need a strong proof to prove which reward function performs better mathematically. Therefore, we conduct a performance evaluation after getting the two average plots for baseline and modified reward function. We calculate AUC for each of the reward function and evaluate the performance.

## 5.2 Type-I and Type-II Comparison

In this section we compare baseline and modified reward functions with each other. Fig. 19 shows initial results of our training. The Tensorboard does not show the legend for each curve, so for better understanding, we use Table 5. It summarizes the experiments

we did to find the difference. Orange-colored curve is the baseline model. Green and pink-colored curves are the modified reward function with $k_1 = 1.0$ and $k_2 = 1.5$. The two curves are the same but different trials. Also, grey curve shows the modified reward function with $k_1 = 1.0$ and $k_2 = 1.1$. We expected that the reward function with $k_2 = 1.5$ would perform better than others. It does perform better if we compare the pink curve with the orange (baseline). However, the green curve performs even worse than the grey curve. Additionally, after 3000 steps of training, all the curves tend to merge. It shows that the weights are learned, and the model tries to converge.

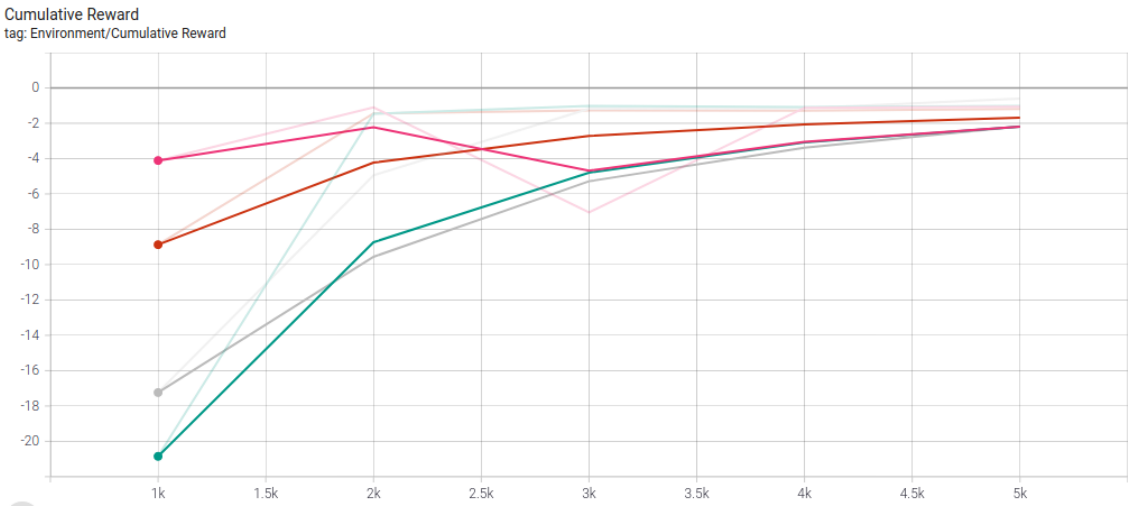


Fig. 19. Trial-1 of experiments: Cumulative Reward.

TABLE 5

Conducted Experiments

| Type | Color | $k_1$ | $k_2$ |
|---|---|---|---|
| Original | Orange | $N/A$ | $N/A$ |
| Modified-1 | Pink, Green | 1.0 | 1.5 |
| Modified-2 | Grey | 1.0 | 1.1 |

For comparison and evaluation, we trained each reward function twice. Fig. 20 shows our second trial of experiments. This figure has many curves, therefore, for easy interpretation we assign a same color arrow to a particular function. From Fig. 20, we can interpret that there is a neck and neck competition between the original and the modified with $k_2 = 1.5$ reward functions. Still, we are unsure about overall performance of the reward functions.

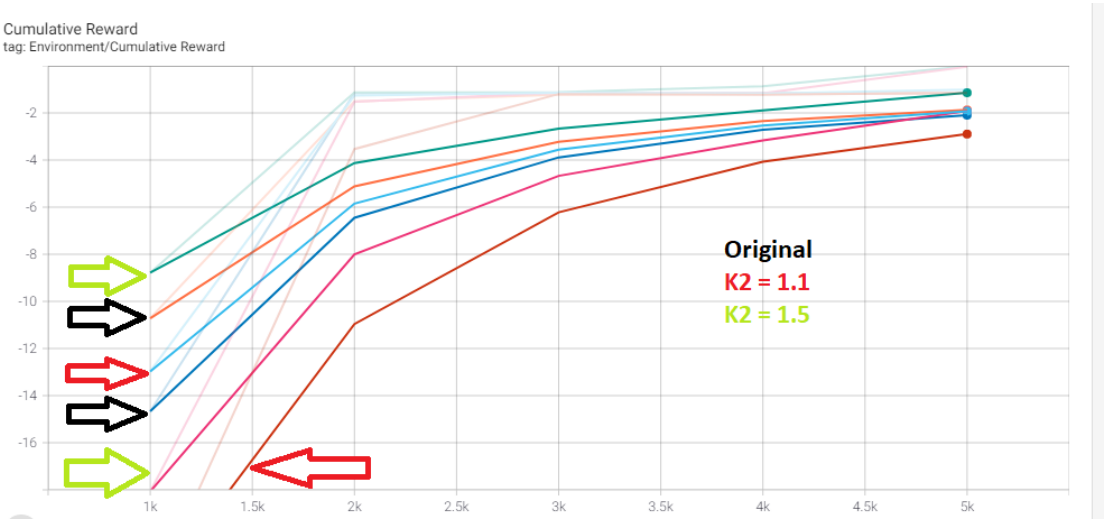


Fig. 20. Trial-2 of the experiments: Cumulative Reward.

Now, we compare episode lengths of each reward function for different trails. Fig. 21 and Fig. 22 are graphs showing the length of episodes for Trial-1 and Trial-2 respectively.

Color codes for episode lenght graphs are same as cumulative reward graphs. We can note that, the episode length is decreasing as the number of steps increases. The episode length is not our main focus. We care about techniques and values to maximize the reward function more.

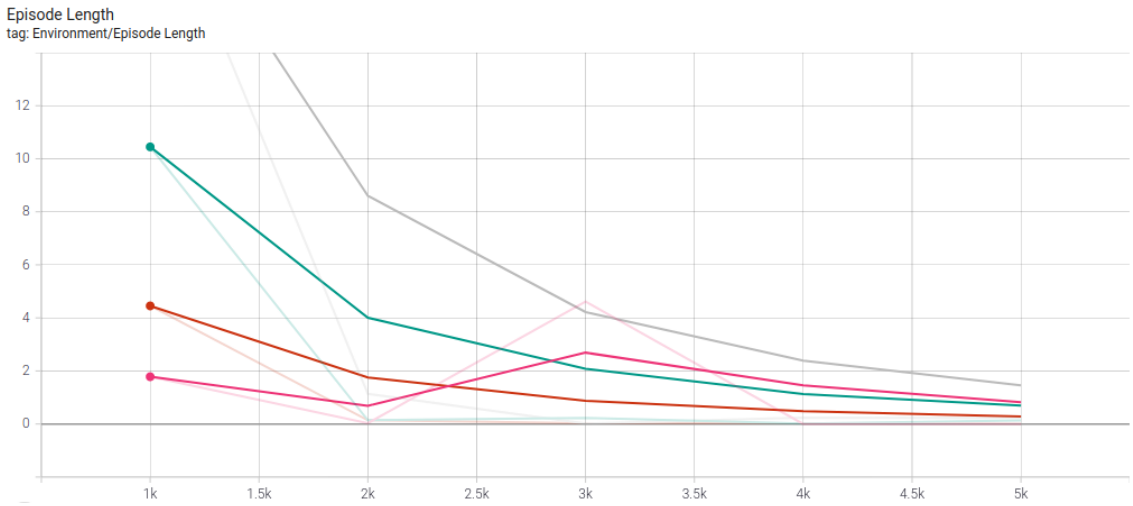


Fig. 21. Trial-1 of the experiments: Episode length.

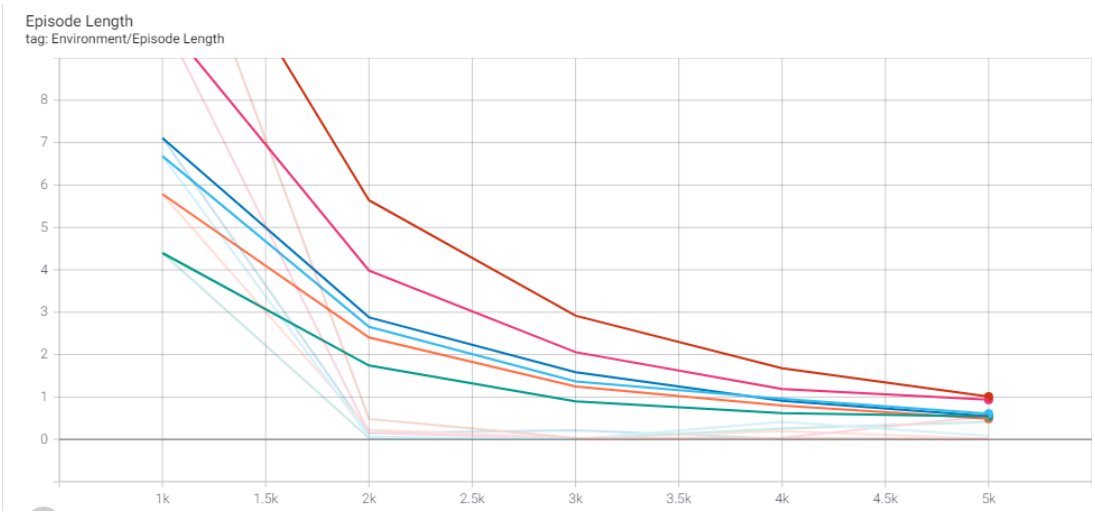


Fig. 22. Trial-2 of the experiments: Episode length.

## 5.3 Performance Evaluation

To evaluate an average performance and come up with a result, we performed training for original and modified $K_2$ functions ten times as discussed in the last section. We examine the performance by plotting ten different curves and one average resultant curve for each reward function.

An average reward function curve will look like Fig. 23. The figure shows an increasing curve. The curve has two different regions; one on the negative and the other on the positive side. A cross-over point divides these regions. We assume the left-hand side of the cross-over point as Area-1 as negative reward region $\Omega_N$. The region on the right-hand side is Area-2 or positive reward region $\Omega_P$. Thus, we expect a smaller negative region or smaller value of $\Omega_N$ and a more significant positive region or immense value of $\Omega_P$.

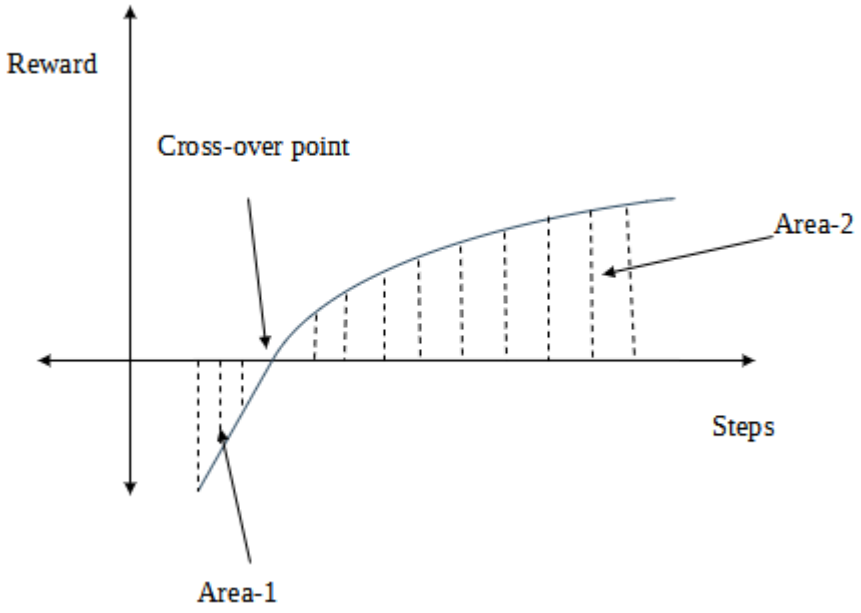


Fig. 23. Performance evaluation curve

We plot two average graphs for baseline as well as $K_2$ rewards. Then, we measure AUC for both the curves and compare it. Naturally, the graph having the larger area will have better performance [38]. Therefore, computationally:

$$\int_{\Omega_N} R(k; s_k, a_k)\, dk + \int_{\Omega_P} R(k; s_k, a_k)\, dk \tag{23}$$

Above Equation 23 can also be written as:

$$\int_{\Omega_N} R(k; s_k, a_k)\, dk = \sum_{k=1}^{k_n} R(k; s_k, a_k) \tag{24}$$

and

$$\int_{\Omega_P} R(k; s_k, a_k)\, dk = \sum_{k=kn}^{k} R(k; s_k, a_k) \tag{25}$$

Indices for accumulated negative and positive rewards are:

$$I_N = \sum_{k=1}^{k_n} R(k; s_k, a_k) \tag{26}$$

and

$$I_P = \sum_{k=k_n}^{k} R(k; s_k, a_k) \tag{27}$$

We introduce a second sub-script for both Equations 26 and 27. So, we have $I_{N,K_2}$ as an index covering the negative accumulated reward for the proposed $K_2$ algorithm and $I_{N,B}$ for the index of the accumulated reward for the base-line algorithm. Similarly, we add the sub-scripts for the indices of positive region of both the reward functions. Therefore,

$$\eta_N = \frac{I_{N,K_2}}{I_{N,B}} \tag{28}$$

and

$$\eta_P = \frac{I_{P,K_2}}{I_{P,B}} \tag{29}$$

To compare the performance of our proposed $K_2$ algorithm with the baseline algorithm by the Google DeepMind team, we have the following guideline.

If $\eta_N < 1$, then accumulated negative reward of the proposed $K_2$ algorithm performs better in generating a lesser negative penalty. And if $\eta_P > 1$, then accumulated positive reward of the proposed $K_2$ algorithm performs better in generating more positive rewards.

### 5.4 Integrated Experiments

This section elaborates our experiments with performance evaluation of the baseline and modified reward functions. To brief the experiments again, due to the random behavior of the reward function, we were unable to conclude the difference in the performance of the reward functions through initial experimental data. Therefore, we conduct the training again for ten times from the start till sixty thousand steps. The training configuration file for both the original and the modified reward functions remains the same; that is, all parameters and hyper-parameters remain constant throughout the experiments. The only change we make is in the code in building the reward functions. Fig. 24 and Fig. 25 show the training results for baseline and modified reward functions respectively.

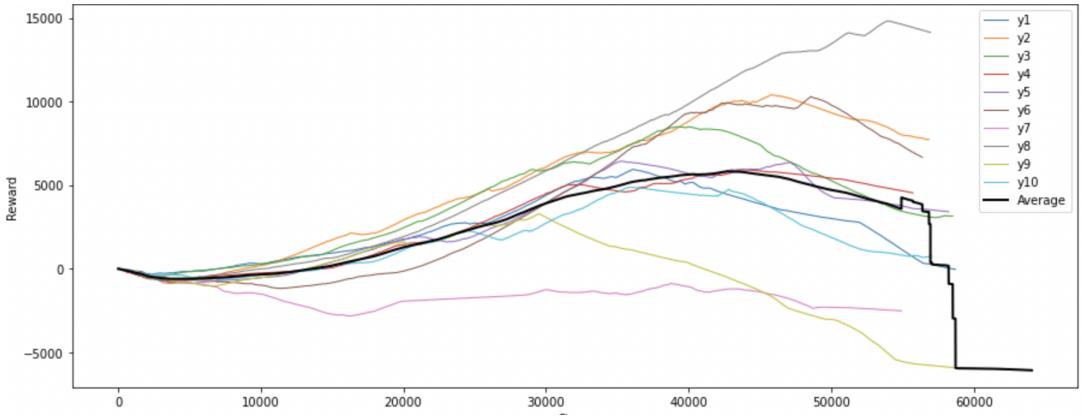


Fig. 24. Baseline algorithm ran for ten different times.

Our initial experimental data was plotted directly on Tensorboard visualization tool without any further modifications. Due to the absence of legend in the graph and difficulty in plotting a customized average graph on the Tensorboard, we use the Python Jupyter notebook environment to plot our final experimental results. We export number of steps and their corresponding reward values to a CSV file for each training experiment. We also export the values of each variable used to calculate the reward at every step. This is a convenient way of tracing back the values of every variable in the reward function. Finally, we plot an accumulative reward value at each step. Clarifications and conclusions drawn from both these graphs are below.

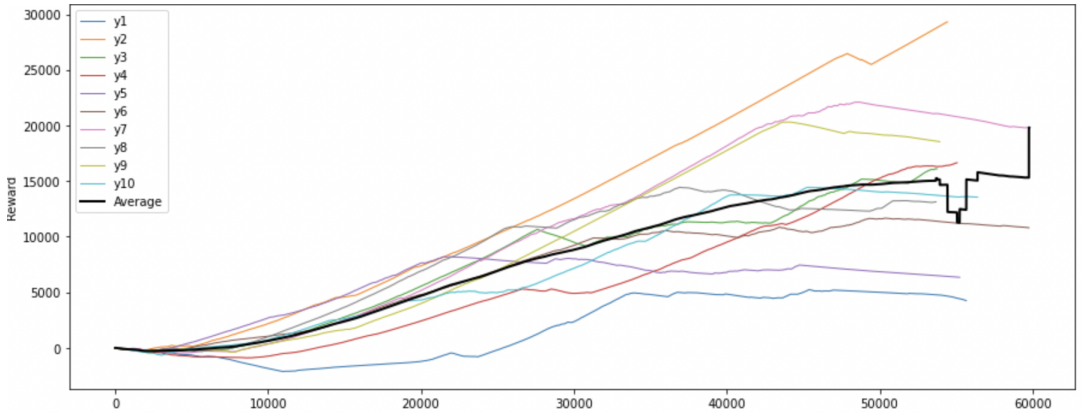


Fig. 25. Modified algorithm ran for ten different times with $K_2 = 1.5$.

Experiments conducted on the baseline reward function is depicted in Fig. 24. The graph shows Steps as an independent variable and Reward as a dependent variable. We can check that there is a total of 11 curves in the figure. First ten graphs with different colors are shown in a legend on the right-most corner are the results of our ten experiments. $11^{th}$ curve from top, is an average curve obtained from these ten experiments. The average curve is a black-colored curve.

Above paragraph discussed simple conclusions from the graph. Now, we dive in-depth, analyzing it closely. As number of steps increases, training improves. The training improvement here means we get a larger reward. Thus, our agent receives a higher reward value as it learns to reach the target object. Also, a higher positive reward means lesser penalty or negative reward. Therefore, the agent is not deviating from required goal and sticks to our pre-defined trajectory of reaching the target object. If we observe the graph closely, we can see that, from step 0 to around 11,000 steps, the agent has a negative reward. The curve is slightly on the negative side of Y-axis. To compare this region with the Fig. 23, which is our performance curve, we get Region-1 or Area-1. Similarly, after 11,000 steps, the curve tends to cross-over to the positive side of Y-axis. The agent starts receiving a positive reward. Thus, for our baseline reward function,

11,000 is a cross-over point. This trend of elevation continues gradually until around 55,000 steps. Again, if we compare this region with the region in Fig. 23, we get Region-2 or Area-2. The Area-2 is the positive area. We expect Area-2 $>$ Area-1. The baseline reward function exceeds our expectation because we can visually note that the positive region is greater than the negative region. We neglect the steps after 55,000 steps as there are some missing training points. There are a few curves with exactly 60,000 training steps and a few curves lesser than that. Due to this, our average curve immediately falls after specific steps in the end. For illustration, brown-colored curve has more than 60,000 training steps, and that trial shows lower positive reward. It makes our average curve fall suddenly. We lack data after around 55k steps; therefore, we neglect that region.

Now, we study modified reward function. Fig. 25 shows experiments conducted using the modified algorithm with $K_1 = 1.0$ and $K_2 = 1.5$. We opt out the function with $K_1 = 1.0$ and $K_2 = 1.1$ as it did not show any better performance than the baseline. This is clearly proved during initial experiments in Fig. 19 and Fig. 20.

The final experiment graph is similar to the previous in terms of the dependent and independent variables. Also, there are a total of 11 curves, including ten different experimental curves and an average curve. The ten different trials of the modified reward function are in various colors. The average curve is shown in black color. If we observe the graphs, the accumulative reward increases as we increase the number of steps of training or train more. As it is an accumulative reward, the values might increase or decrease depending upon the reward value at the current step and at the previous step. The reward gets negative if both these values are negative. In other words, the agent is not able to learn at every step. However, this is not the case for our results. The graph increases gradually at each step that understandably states that more positive rewards get added. Until around 7,000 steps, the graph has negative values of reward. After these 7,000 steps, the curves try to cut off the X-axis and diverts in the positive region of the Y-axis. If we

compare the curves with our performance curve Fig. 23, we have Area-1 from 0 to 7,000 steps and Area-2 from 7000 steps to 55,000 steps. The cross-over point for our modified reward function is at around $7,000^{th}$ step. Our expectation of Area-1 $<$ Area-2 seems to be true for the modified reward function too. Here also, we neglect steps after 55,000 because there are some missing values. We do that to match the steps with our baseline model. A few curves have training values beyond 55,000 until 60,000. We fix the stopping point as 55,000 and we do not care about the values after these many steps.

Our main goal is to compute and distinguish the performance of the baseline and the modified reward functions. Then, after comparing the ten different graphs within the same group, we compare them to derive conclusions visually. Unfortunately, it is questioning to trace 20 curves simultaneously and compare them individually with each other. Therefore, we only use the average graphs from each group to compare the overall performance of Steps vs. Reward.

There are two curves in Fig. 26, namely: baseline reward function and modified reward function. These two curves are average curves of the 10 graphs. Black-colored curve is an average curve of 10 experiments for the baseline reward function, whereas red-colored curve is an average curve of 10 experiments for the modified reward function. We only consider the steps from 0 to 55,000. The above sections already explained the reasons.

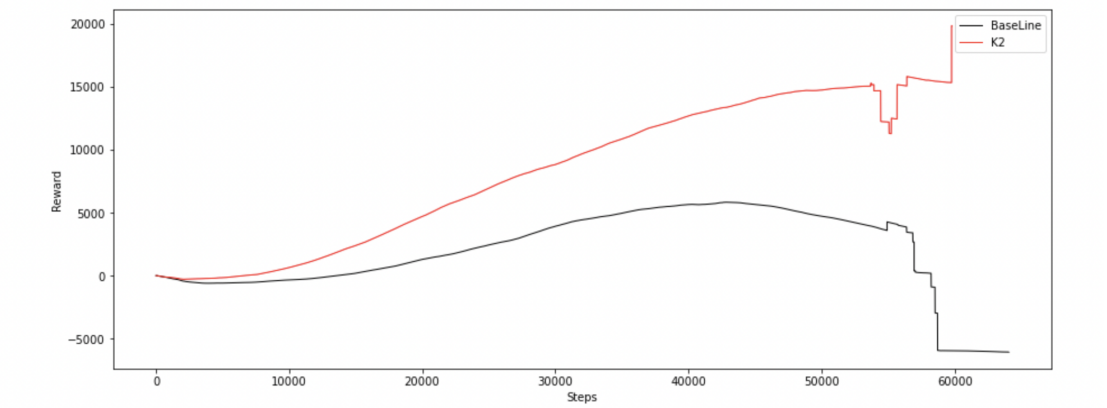


Fig. 26. Average of baseline and modified reward function.

We can see that the modified algorithm is better than the baseline by examining the graph without diving into the details. The baseline curve starts to cross the X-axis at around $11,000^{th}$ step while the modified reward function crosses before it. The cross-over point for modified reward function is approximately at $7,000^{th}$ step. Visibly, we can mark that the Area-1 of baseline reward function is greater than Area-1 of the modified reward function. In contrast, the Area-2 of the modified reward function is greater than Area-2 of the baseline reward function. The modified reward function has an accumulative reward value of 15,000 at 55,000 steps. Conversely, the baseline only reaches 5,000 accumulative reward value at 55,000 steps. Gap between the graphs widens after about 40,000 steps. The modified reward algorithm can be considered as merit-based as it shows a quick increase in the reward values. These interpretations are just visual.

We create stopping criteria in our code and calculate stopping steps at which at least one of the trials starts having missing values. We also precisely get cross-over points for both the reward functions. Fig. 27 shows the baseline reward function after eliminating the missing values. Fig. 28 shows the modified reward function after eliminating the missing values. Now, we can explicitly note that the average graphs do not decline drastically as before.

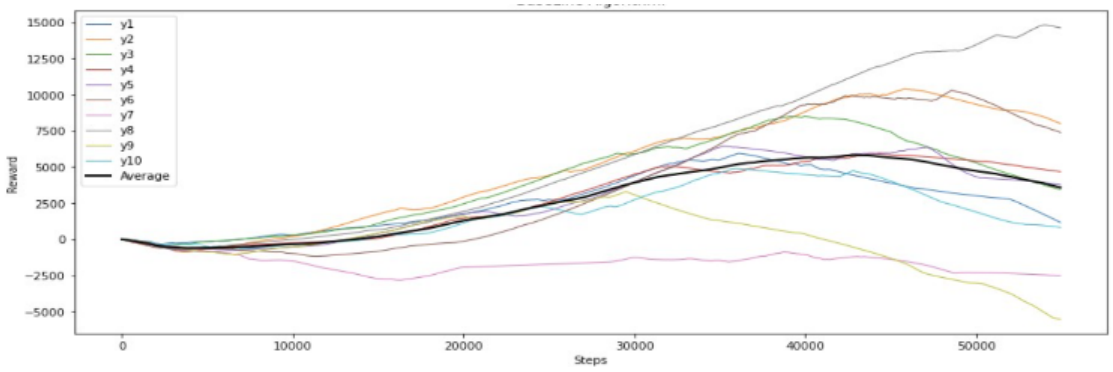


Fig. 27. Baseline reward function after eliminating missing values.

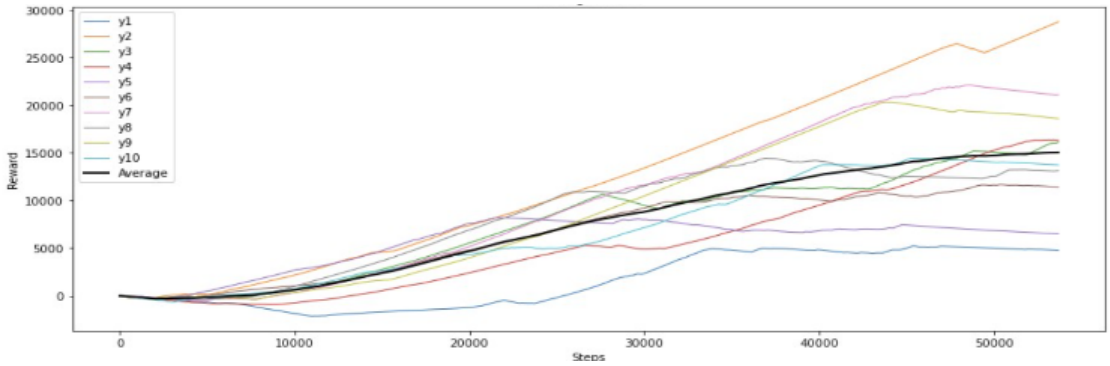


Fig. 28. Modified reward function after eliminating missing values.

Now, we combine both of these graphs and get a plot in Fig. 29. The plots show average of both baseline as well as modified reward functions after deducting the missing values. The graph also shows the cross-over points precisely. The baseline cross-over point is at $13,635^{th}$ step, and the modified cross-over point is at $6529^{th}$ step. Corresponding reward values for the baseline cross-over step is 0.08, and for the modified cross-over step is 0.225. Thus, the cross-over for the modified reward function happens earlier than the baseline reward function. Additionally, the value of the reward is also more remarkable for the modified reward function at the cross-over point.

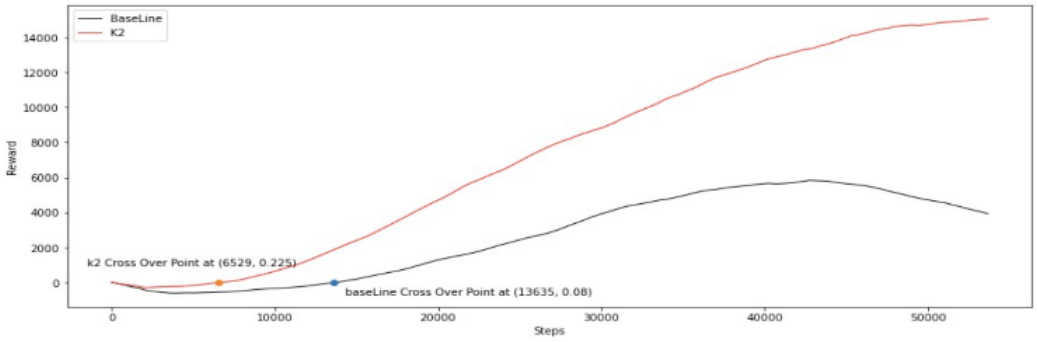


Fig. 29. Average reward functions after eliminating missing values.

We can prove that the modified reward function is better than the baseline using mathematical calculations too. We show performance evaluation by calculating $\eta$ values explained in above sections.

From Equation 23, we can form an equation for original reward function as:

$$\int_{k=1}^{k=13,635} R(k;s_k,a_k)\,dk + \int_{k=13,636}^{k=53,689} R(k;s_k,a_k)\,dk \tag{30}$$

Similarly, we have modified reward function as:

$$\int_{k=1}^{k=6,529} R(k;s_k,a_k)\,dk + \int_{k=6,530}^{k=53,689} R(k;s_k,a_k)\,dk \tag{31}$$

Equation 30 can also be written as:

$$\int_{k=1}^{k=13,635} R(k;s_k,a_k)\,dk = \sum_{k=1}^{13,635} R(k;s_k,a_k) \tag{32}$$

and

$$\int_{k=13,636}^{k=53,689} R(k;s_k,a_k)\,dk = \sum_{k=13,636}^{53,689} R(k;s_k,a_k). \tag{33}$$

Also, Equation 31 can be written as:

$$\int_{k=1}^{k=6,529} R(k;s_k,a_k)\,dk = \sum_{k=1}^{6,529} R(k;s_k,a_k) \tag{34}$$

and

$$\int_{k=6,530}^{k=53,689} R(k;s_k,a_k)\,dk = \sum_{k=6,530}^{53,689} R(k;s_k,a_k). \tag{35}$$

We do the computation by integration or discrete summation and find AUC using Python 3 code on the Jupyter Notebook environment. Therefore, from Equation 26, index of the negative region of the baseline reward function becomes:

$$I_{N,B} = \sum_{k=1}^{11,000} R(k;s_k,a_k) \tag{36}$$

And, index of the positive region of the baseline reward function becomes as follows:

$$I_{P,B} = \sum_{k=11,001}^{55,000} R(k;s_k,a_k) \tag{37}$$

The indices for modified reward function are also shown below:

$$I_{N,K_2} = \sum_{k=1}^{7,000} R(k;s_k,a_k) \tag{38}$$

$$I_{P,K_2} = \sum_{k=7,001}^{55,000} R(k;s_k,a_k) \tag{39}$$

Further, we calculate the $\eta_N$ and $\eta_P$ values. We re-write Equation 28 and Equation 29 below:

$$\eta_N = \frac{I_{N,K_2}}{I_{N,B}} \tag{40}$$

and

$$\eta_P = \frac{I_{P,K_2}}{I_{P,B}} \tag{41}$$

Some important values required for calculation of $\eta_N$ and $\eta_P$ are in Table 6. The table consists values of $I_{N,B}$ , $I_{N,K_2}$ , $I_{P,B}$ , $I_{P,K_2}$ for the calculation.

TABLE 6

Important Values from Average Curves

| **Category** | **Baseline Reward Function** | **Modified Reward Function** |
|---|---|---|
| Cross-over point | $13,635^{th}$ step | $6529^{th}$ step |
| $I_N$ | $-5.21e^{+6}$ | $-1.1e^{+6}$ |
| $I_P$ | $1.53e^{+8}$ | $3.97e^{+8}$ |

Substituting these values in Equation 28 and Equation 29 we get:

$$\eta_N = \frac{I_{N,K_2}}{I_{N,B}} = 0.212 \tag{42}$$

and

$$\eta_P = \frac{I_{P,K_2}}{I_{P,B}} = 2.595 \tag{43}$$

Our assumption of getting $\eta_P > \eta_N$ proves to study the above results genuinely. The value of $\eta_N$ is less than 1.0, which means the value of $I_{N,B}$ is high or $I_{N,K_2}$ is low. Similarly, for $\eta_P$. We get 259.5% of improvement in the positive region of the modified reward function.

# 6 CONCLUSIONS

In this research, we investigate and design an accelerated reward function for robotics application of reaching the target. The robotics application can be programmed easily using any coding language. However, for complex tasks in which the programmer cannot predict all the states and actions of the robot, the programming becomes difficult. Hence, there arises a need to have a technique to make robots learn tasks even if they encounter an uncommon situation or state. DRL allows us to train the robots/agents by exploration and exploitation process. We put the robot in a situation to interact with the environment and learn from its mistakes by providing feedback. Feedback plays a vital role in making the agent aware of following a desired goal or not. If the agent deviates from a particular behavior, there is a harsh penalty and if the agent follows the desired trajectory, then it gets a reward. The robot collects the feedback and accumulates it. Thus, it tries to maximize a long-term reward. By keeping this in mind, we modify the reward function to get an accelerated and highest long-term reward.

We show the utilization of Google DeepMind's SAC algorithm in a continuous control application. We have different motor movements for reaching the desired target. We choose the Turin robot, which is a 6 DoF industrial robot. Each axis has a different movement in terms of angle or torque. Thus, a combination of motors in the robotic arm movement becomes an infinite set of possibilities. Therefore, the traditional discrete action-based policy algorithms such as DDPG are not applicable for more complex tasks. We saw that the SAC algorithm has its benefits if compared to the PPO algorithm.

DRL technique helps a robot to learn from mistakes and experiences. In this process, the robot might encounter some situations that can damage the system, which is unacceptable. To avoid hazardous situations, we require a simulation platform for training and allowing the robot to make mistakes. We utilize the Unity 3D simulation platform for learning process. Unity proves to be a better option than any other simulation

environment. A high-end GUI and complex yet simple to use physics make Unity 3D a general platform for researchers.

We take a baseline reward function developed by an author using Google DeepMind's policy and Unity 3D environment. We modify the reward function in such a way that we get an accelerated output. First, we clearly define the relationship between reward and penalty with the distance of the target from the gripper. The task of reaching the target is related to the distance; therefore, we use distance as one of the most decisive parameters for calculating the reward. Then, to accelerate the process, we introduce two gain factors in the equations.

We perform experiments by changing values of gain factors. The gain factor $k_1$ is related to the penalty, while $k_2$ is related to the reward. The penalty should be less than the reward to get a greater reward. So, we keep the penalty gain factor constant and just change the reward gain factor.

For verification and testing, we undergo the training for ten different times. First, we plot an average graph using the ten different value functions. After plotting both graphs, we find AUC. Next, we calculate two different index values for each side of the curve—the negative index from the penalty and the positive index from the rewards. We then calculate $\eta$ and decide which function performs better based on the value. By going through this process, we conclude that the merit-based modified reward function performs well. We noted about 259.5 percent of improvement when compared to the baseline reward function.

## Literature Cited


[1] Andrew Kusiak, "Smart manufacturing," *International Journal of Production Research*, pp. 508–517, 2018.

[2] Kravets and A. G, *Robotics: Industry 4.0 Issues& New Intelligent Control Paradigms*. Springer, 2020. ISBN: 3030378403.

[3] R. Y. Zhong, X. Xu, E. Klotz, and S. T. Newman, "Intelligent manufacturing in the context of industry 4.0: A review," *Engineering*, vol. 3, no. 1, pp. 616–630, 2017.

[4] S. Ritter, "Meta-reinforcement learning with episodic recall: An integrative theory of reward-driven learning," *ProQuest Dissertations and Theses*, p. 160, 2019. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2020-11-18.

[5] M. Volodymyr, K. Koray, S. David, R. A. A, V. Joel, B. M. G, G. Alex, R. Martin, F. A. K, O. Georg, P. Stig, B. Charles, S. Amir, A. Ioannis, K. Helen, K. Dharshan, W. Daan, L. Shane, and H. Demis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, p. 529–533, 2015.

[6] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," April 2016. https://arxiv.org/abs/1504.00702.

[7] H. Moss, "Genes, affect, and reason: Why autonomous robot intelligence will be nothing like human intelligence," *Techné: Research in Philosophy and Technology*, vol. 20, pp. 1–15, 2015.

[8] M. P. Deisenroth, G. Neumann, J. Peters, *et al.*, "A survey on policy search for robotics," *Foundations and trends in Robotics*, vol. 2, no. 1-2, pp. 388–403, 2013.

[9] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," January 2019. https://arxiv.org/abs/1812.05905.

[10] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," 2016.

[11] R. K, "How to train your robot arm?," August 2020. https://medium.com/xrpractices/how-to-train-your-robot-arm-fbf5dcd807e1.

[12] M. Wiering and M. van Otterlo, *Reinforcement Learning*, vol. 12. Springer, Berlin, Heidelberg, 2012.

[13] T. H. Cormen, C. E. Leiserson, R. Rivest, and C. Stein, *Introduction To Algorithms*. MIT Press, December 2000.

[14] C. Boutilier, T. Dean, and S. Hanks, "Decision-Theoretic Planning: Structural Assumptions and Computational Leverage," *Artificial Intelligence Research 11*, July 1999.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[16] A. Iriondo, E. Lazkano, L. Susperregi, J. Urain, A. Fernandez, and J. Molina, "Pick and place operations in logistics using a mobile manipulator controlled with deep reinforcement learning," *Applied Sciences*, vol. 9, no. 2, 2019.

[17] *AAMAS '20: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, (Richland, SC), International Foundation for Autonomous Agents and Multiagent Systems, 2020.

[18] Z. Yi, Q. Yu, Y. Yichen, H. Haoyuan, and X. Yinghui, "Learning to cooperate: Application of deep reinforcement learning for online agv path finding," *Proceedings of the 19th International Conference on autonomous agents and multiagent systems*, pp. 2077–2079, May 2020.

[19] D. Schwab and S. Ray, "Offline reinforcement learning with task hierarchies," *Machine learning*, vol. 106, no. 9, pp. 1569–1598, 2017.

[20] Y. Yue, Y. Tang, M. Yin, and M. Zhou, "Discrete action on-policy learning with action-value critic," February 2020. https://arxiv.org/abs/2002.03534.

[21] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin, "Deep reinforcement learning in large discrete action spaces," 2016.

[22] J. Stopforth and D. Moodley, "Continuous versus discrete action spaces for deep reinforcement learning," 2019. http://ceur-ws.org/Vol-2540/FAIR2019_paper_47.pdf.

[23] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," July 2019. https://arxiv.org/abs/1509.02971.

[24] W. Lötzsch, "Using deep reinforcement learning for the continuous control of robotic arms," October 2018. https://arxiv.org/abs/1810.06746.

[25] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," August 2017. https://arxiv.org/abs/1707.06347.

[26] J. Peters, "Policy gradient methods," *Scholarpedia journal*, vol. 5, no. 11, p. 3698, 2010.

[27] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A general platform for intelligent agents," May 2020. https://arxiv.org/abs/1809.02627.

[28] U. Team, "Unity-robotics-hub," 2021. https://github.com/Unity-Technologies/Unity-Robotics-Hub.

[29] MPinol, "Ros-unity Integration," 2021. https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/tutorials/pick_and_place/README.md.

[30] Niryo, "Ned," 2021. https://niryo.com/product/ned/.

[31] Playfish, "Urdf," 2019. http://wiki.ros.org/urdf.

[32] ROS.org, "Moveit," 2021. https://moveit.ros.org/.

[33] S. Annambhotla, C. Romero, and A. Thaman, "Synthetic data: Simulating myriad possibilities to train robust machine learning models," May 2020. https://blogs.unity3d.com/2020/05/01/synthetic-data-simulating-myriad-possibilities-to-train-robust-machine-learning-models/.

[34] Unity, "Automotive, transportation and manufacturing," 2021. https://unity.com/solutions/automotive-transportation-manufacturing.

[35] A. Franceschetti, E. Tosello, N. Castaman, and S. Ghidoni, "Robotic arm control and task training through deep reinforcement learning," 2020.

[36] H. Li, “Deep reinforcement learning for robotics arm control part i,” February 2021. https://github.com/hualili/robotics-open_abb/blob/master/fd100/105b-fd100-robot-unity-hl-2021-3-14.pdf.

[37] C. V. . H. L. . Y. Yakuwa, “Readme for robot arm ml agent unity,” April 2021. https://github.com/hualili/robotics-open_abb/blob/master/fd100/105e-readme-Unity-Robot-Arm-ML-Agents-CV-HL-YY-2021-4-9.pdf.

[38] H. Li, “Sac soft actor critic algorithm overview,” June 2021. https://github.com/hualili/robotics-open_abb/blob/master/fd100/105e-sac-hl-2021-6-3.pdf.

[39] E. Teng, “Training your agents 7 times faster with ml-agents,” November 2019. https://blog.unity.com/technology/training-your-agents-7-times-faster-with-ml-agents.

[40] H. Li, “Turnin user guide,” April 2021. https://github.com/hualili/roboticsopen_abb/blob/master/fd100/100-Turin%20Smart%20Robot%20General%20Operation%20Manual%20V2.0.20214-18.pdf.pdf.

[41] C. Vang, “Robotcontroller baseline,” June 2021. https://github.com/hualili/robotics-open_abb/blob/master/fd100/105g-1RobotControllerAgent-1Base.cs.

[42] C. Vang, “Robotcontroller time index,” June 2021. https://github.com/hualili/robotics-open_abb/blob/master/fd100/105g-2RobotControllerAgent-2timeInd.cs.

[43] C. Vang, “Robotcontroller time index,” June 2021. https://github.com/hualili/robotics-open_abb/blob/master/fd100/105g-3RobotControllerAgent-3K2.cs.

[44] N. Vadher, “Performance evaluation,” June 2021. https://github.com/hualili/robotics-open_abb/blob/master/fd100/105g%23105-4-ReadMe-K2-Base-Average-Eta-NV-2021-06-22.pdf.

## Appendix A

### CONFIGURATION FILE

The configuration file is a requirement for the training of Deep Reinforcement Learning on the Unity platform. The file's extension is a .yaml file. YAML stands for YAML Ain't Markup Language. The configuration file should be saved in the project folder for the $ML-Agents$ to access it. Typically it is stored in the main project folder $RobotArmMLAgentUnity > trainer\_config.yaml$. The file initializes the parameters and hyper-parameters required to start, process, and end the agent's training using the ML-Agents Unity package. Some parameters are common for all the training. However, some parameters are specific to the type of algorithm or trainer we utilize to train our agent.

Additionally, there is some ambiguity in terms of Neural Network architecture and its hyper-parameters. These parameters can change as per the complexity of the tasks. Thus, we recommend training with different values of the hyper-parameters before finalizing them. The footnote below shows the configuration file for our application.

```
//-----------------------------------------------------------/
// program    : trainer-configuration.yaml                   /
// coded by   : Unity                                        /
// updated by : Shifa Shaikh, Chee Vang                      /
// note       : Comments by SS                               /
//-----------------------------------------------------------/
behaviors:
    default:
        trainer_type: sac # algorithm selection
        hyperparameters: # Neural Network Training
            batch_size: 256 # training batch-size
            buffer_size: 2560 # buffer-size for collecting experiences
            learning_rate_schedule: linear # linear decay for stable
                                           #convergence
            learning_rate: 3.0e-4 # learning rate
        network_settings: # Neural Network architecture
            hidden_units: 256 # No. of hidden neurons
            normalize: false # normalize the vector input
            num_layers: 3 # No. of hidden layers
            vis_encode_type: simple # encoding of visual
                             #observations
            memory:
                memory_size: 256 # to save the previous experience
```

```
            sequence_length: 256 # sequence of experience while
            #training
    max_steps: 10.0e5 # Training steps
    summary_freq: 10000 # printing the status on console
    reward_signals: # DRL parameters
        extrinsic:
            strength: 1.0
            gamma: 0.99
Robotarm:
    trainer_type: sac
    hyperparameters:
        batch_size: 256
        buffer_size: 2560
    network_settings:
        hidden_units: 256
        num_layers: 3
    max_steps: 5.0e3
    time_horizon: 128
    summary_freq: 1000
```

## Appendix B

### BASELINE ALGORITHM WITH TIME-INDEX

The listing below shows the C sharp code for the baseline reward function. This script is the running script for the agent [41]. It should save in the *Scripts* folder of the project. The script can be found in the file structure as *RobotArmMLAgentUnity* > *Assets* > *Scripts* > *RobotControllerAgent.cs*. This script is for the baseline algorithm. We can differentiate the scripts by looking at line number 293 in the below code. The line declares a variable *reward_flag* as a Boolean data type. We assign *False* to the variable.Therefore, the original algorithm will implement the, *else* statement from the line numbers 326 to 353. For more information, please check the code below and the corresponding comments for understanding [42].

```
1  //-------------------------------------------------------------/
2  // program     : RobotControllerAgent_piece-chee-2021-5-14.cv  /
3  // coded by    : Unity                                         /
4  // updated by  : Chee Vang, HL                                 /
5  // note        : 1. re-wrote reward algorithm as picewise with /
6  //                  two line segments & k1,k2 gain factor      /
7  //-------------------------------------------------------------/
8  using System;
9  using System.Data.Common;
10 using System.Linq;
11 using Unity.MLAgents;
12 using Unity.MLAgents.Sensors;
13 using UnityEngine;
14 using Random = UnityEngine.Random;
15 // 2021-5-14: File I/O
16 using System.IO;
17 using System.Text;
18 // 2021-5-17: List/Queues
19 using System.Collections.Generic;
20
```

```
21 //------------------------------------------------------------/
22 public class RobotControllerAgent : Agent
23 {
24     [SerializeField]
25     private GameObject[] armAxes;
26     [SerializeField]
27     private GameObject endEffector;
28
29     public bool trainingMode;
30     private bool inFrontOfComponent = false;
31     public GameObject nearestComponent;
32     private Ray rayToTest = new Ray();
33     private float[] angles = new float[5];
34     private KinematicsCalculator calculator;
35     private float beginDistance;
36     private float prevBest;
37     bool  inrange2 = false, inrange3 = false, inrange4 = false, inrange5 = false
           ;
38     private float baseAngle;
39     private const float stepPenalty = -0.0001f;
40     private string path = @"/home/chee/Documents/RobotArmMLAgentUnity/distance.
           csv";
41     // CV 2021-5-14: File path for CSV file
42
43     private int epi_index = 1;
44     // CV 2021-5-14: index used to determine number of times OnEpisodeBegin was
           called
45     private int t;
46     // CV 2021-5-17: a time index
47     private int buffLength = 3;
48     //1024;
49     // Queue
50     private Queue<float> deltaDistBuffer = new Queue<float>();
51     // CV 2021-5-20: Buffer for delta_distance
52     private Queue<float> distanceBuffer = new Queue<float>();
53     // CV 2021-5-20: Buffer for distance
```

```
54      private Queue<float> prevBestBuffer = new Queue<float>();
55      // CV 2021-5-20: Buffer for prevBest
56      private Queue<float> tBuffer = new Queue<float>();
57      // CV 2021-5-20: Buffer for time index t
58
59      //------------------------------------------------------------/
60      // CV MonoBehaviour::Start()
61      // called on frame when script is enabled just before any update methods
62      //      are called the first time
63      //- referece: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.
            html
64
65
66      private void Start()
67      {
68
69      }
70
71      //------------------------------------------------------------/
72      // Agent::Initialize ()
73      //   - Perfroms a one-time initialization or set up of the agent instance
74      //   - MLAgents API referece:
75  //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.
        Agent.html#Unity_MLAgents_Agent_Initialize
76
77
78      public override void Initialize()
79      {
80          ResetAllAxis();
81          // 2021-6-8: fixed starting endeffector position
82          MoveToSafeUserPreDefinedPostion();
83          //MoveToSafeRandomPosition();
84          if (!trainingMode) MaxStep = 0;
85          // CV, SS 2021-5-26: Label the columns in CSV file
86
87          using(StreamWriter writer = new StreamWriter(path,false)) {
```

```
88              writer.WriteLine("t" + "," +
89                      "beginDistance" + "," +
90                      "prevBest" + "," +
91                      "distance" + "," +
92                      "delta_distance" + "," +
93                      "reward" + "," +
94                      "target x" + "," +
95                      "target y" + "," +
96                      "target z" + "," +
97                      "endEff x " + "," +
98                      "endEff y " + "," +
99                      "endEff z ");
100             writer.Close();
101         }
102
103     }
104
105     //-----------------------------------------------------------/
106     // ResetAllAsix()
107     //   - resets all axis to zero
108     //   - function called in
109     //         1) Initialized()
110     //         2) OnEpisodeBegin()
111
112     private void ResetAllAxis()
113     {
114         armAxes.All(c =>
115         {
116             c.transform.localRotation =  Quaternion.Euler(0f, 0f, 0f);
117             return true;
118         });
119     }
120
121     //-----------------------------------------------------------/
122     // Agent::OnEpisodeBegin()
123     //   - Set up an agent instance to the start of an episode
```

```
124     //    - Resets agent and environment to their starting position where
125     //      reset values are randomized
126     //    - MLAgents API reference:
127     //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.
            MLAgents.Agent.html#Unity_MLAgents_Agent_OnEpisodeBegin
128     //    - MLAgents Github reference:
129     //https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-
            Environment-Design-Agents.md
130
131     public override void OnEpisodeBegin()
132     {
133         if(trainingMode)
134             ResetAllAxis();
135
136         // 2021-6-8: fixed starting endeffector position
137         MoveToSafeUserPreDefinedPostion();
138         //MoveToSafeRandomPosition();
139         UpdateNearestComponent();
140
141         // CV 2021-5-17: Resets time index at start of new episode
142         //    - time_index is updated at each step and resetted at the
143         //      start of a new episode
144
145         t = 0;
146
147         // CV 2021-5-14: Notifies start of a new episode in the CSV file
148         //    - epi_index = number of times this OnEpisdoeBegin() is called
149         //    - CompletedEpisodes = MLAgents C# property that returns number
150         //      of completed episodes
151         //     MLAgents API reference:
152         //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.
              MLAgents.Agent.html
153         /*using(StreamWriter writer = new StreamWriter(path,true)) {
154        //writer.WriteLine("--- Start New Episode #" + epi_index + " --- Agent Ep:
              " + CompletedEpisodes);
155             writer.Close();
```

```
156         }
157         epi_index++;*/
158
159         // CV 2021-6-2: Prints the initial data into CSV file
160
161         float distance = Vector3.Distance(endEffector.transform.TransformPoint(
                Vector3.zero),
162             nearestComponent.transform.position);
163         float delta_distance = distance - prevBest; // CV 2021-5-19: delta
                distance
164         float reward = 0;
165         using(StreamWriter writer = new StreamWriter(path,true)) {
166             writer.WriteLine( t + "," +
167                               beginDistance + "," +
168                               prevBest + "," +
169                               distance + "," +
170                               delta_distance + "," +
171                               reward + "," +
172                               nearestComponent.transform.position.x + "," +
173                               nearestComponent.transform.position.y + "," +
174                               nearestComponent.transform.position.z + "," +
175                               endEffector.transform.position.x + "," +
176                               endEffector.transform.position.y + "," +
177                               endEffector.transform.position.z );
178             writer.Close();
179         }
180         t++;
181     }
182
183     //-----------------------------------------------------------/
184     // UpdateNearestComponent()
185     //   - Original author's function to update the target ("nearestComponent")
186     //     position to a random, reachable location
187     //   - function is called when
188     //       1) OnEpisodeBegin() is called
189     //       2) end effector reaches the target (Note: orginal code does
```

```
190     //            not end episode when end effector reaches target, see
191     //            JackpotRewar() for more details)
192     //    - computes "beginDistance" and initializes "prevBest" with that value
193     //    - checks that base angle (the whole robot arm, in Unity it is the
            prefab
194     //      "RobotwithColliders" in the hierarchy) is not negative value
195     //        ---> will need to confirm with Shifa
196
197
198     private void UpdateNearestComponent()
199     {
200         /*if (trainingMode)
201         {
202             inFrontOfComponent = UnityEngine.Random.value > 0.5f;
203         }
204         if(!inFrontOfComponent)
205             nearestComponent.transform.position = transform.position +
206 new Vector3(Random.Range(0.3f,0.6f),Random.Range(0.1f,0.3f), Random.Range(0.3f
        ,0.6f));
207         else
208         {
209             nearestComponent.transform.position = endEffector.transform.
                    TransformPoint(Vector3.zero) +
210  new Vector3(Random.Range(0.01f,0.15f),Random.Range(0.01f,0.15f), Random.Range
        (0.01f,0.15f));
211         }*/
212
213         // CV 2021-5-27: fixed target location
214
215         nearestComponent.transform.position = transform.position + new Vector3
                (0.45f,0.2f,0.45f);
216
217         beginDistance = Vector3.Distance(endEffector.transform.TransformPoint(
                Vector3.zero),
218  nearestComponent.transform.position);
219         prevBest = beginDistance;
```

```
220
221         baseAngle = Mathf.Atan2( transform.position.x - nearestComponent.
                transform.position.x,
222     transform.position.z - nearestComponent.transform.position.z) * Mathf.Rad2Deg;
223         if (baseAngle < 0) baseAngle = baseAngle + 360f;
224     }
225
226     //---------------------------------------------------------/
227     // Agent::CollectObservations()
228     //   - MLAgents C# function to collect vector observations of the agents for
            the step
229     //     that describes the current environment from the perspective of the
            agent
230     //   - is called every step when agent requests a decision
231     //   - MLAgents API reference:
232     //   - MLAgents Github reference:
233     //https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-
            Environment-Design-Agents.md
234     /// <summary>
235     /// Markov Decision Process - Observes state for the current time step
236     /// </summary>
237     /// <param name="sensor"></param>
238
239
240     public override void CollectObservations(VectorSensor sensor)
241     {
242         sensor.AddObservation(angles);
243         sensor.AddObservation(transform.position.normalized);
244         sensor.AddObservation(nearestComponent.transform.position.normalized);
245         sensor.AddObservation(endEffector.transform.TransformPoint(Vector3.zero).
                normalized);
246         Vector3 toComponent = (nearestComponent.transform.position - endEffector.
                transform.TransformPoint(Vector3.zero));
247         sensor.AddObservation(toComponent.normalized);
248         sensor.AddObservation(Vector3.Distance(nearestComponent.transform.
                position,endEffector.transform.TransformPoint(Vector3.zero)));
```

```
249         sensor.AddObservation(StepCount / 5000);
250     }
251
252     //--------------------------------------------------------------/
253     // Agent::OnActionReceived()
254     //    - MLAgents C# function used to specify the agent behaviour at every
255     //      step based on the provided action
256     //    - is called every time the agent receives an action to take
257     //    - common to assign rewards in here
258     //    - MLAgents API reference:
259     //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.
            MLAgents.Agent.html#
            Unity_MLAgents_Agent_OnActionReceived_System_Single___
260     //    - MLAgents Github reference:
261     //https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-
            Environment-Design-Agents.md
262
263
264     public override void OnActionReceived(float[] vectorAction)
265     {
266         angles = vectorAction;
267         if (trainingMode)
268         {
269             // Translate the floating point actions into Degrees of rotation for
                    each axis
270
271             armAxes[0].transform.localRotation =
272                 Quaternion.AngleAxis(angles[0] * 180f, armAxes[0].GetComponent<Axis
                        >().rotationAxis);
273             armAxes[1].transform.localRotation =
274                 Quaternion.AngleAxis(angles[1] * 90f, armAxes[1].GetComponent<Axis
                        >().rotationAxis);
275             armAxes[2].transform.localRotation =
276                 Quaternion.AngleAxis(angles[2] * 180f, armAxes[2].GetComponent<Axis
                        >().rotationAxis);
277             armAxes[3].transform.localRotation =
```

```
278             Quaternion.AngleAxis(angles[3] * 90f, armAxes[3].GetComponent<Axis
                    >().rotationAxis);
279         armAxes[4].transform.localRotation =
280             Quaternion.AngleAxis(angles[4] * 90f, armAxes[4].GetComponent<Axis
                    >().rotationAxis);
281
282         float distance = Vector3.Distance(endEffector.transform.TransformPoint
                (Vector3.zero),
283             nearestComponent.transform.position);
284         float diff = beginDistance - distance;
285         float delta_distance = distance - prevBest; // CV 2021-5-19: delta
                distance
286         float delta_R = beginDistance-prevBest; // CV 2021-5-19:
287
288         float k1 = 1.0f;
289     // 2021-5-14 CV,HL: gain factor for Line 1
290         float k2 = 1.5f;
291         // 2021-5-14 CV,HL: gain factor for Line 2
292         bool reward_flag = false;
293     // 2021-5-17 CV: flag for using original or our reward function
294         float reward;
295
296         // CV,SS 2021-5-27: updates the queues with latest 1024 data
297
298         if(deltaDistBuffer.Count >= buffLength)
299         {
300             deltaDistBuffer.Dequeue();
301             distanceBuffer.Dequeue();
302             prevBestBuffer.Dequeue();
303             tBuffer.Dequeue();
304         }
305         deltaDistBuffer.Enqueue(delta_distance);
306         distanceBuffer.Enqueue(distance);
307         prevBestBuffer.Enqueue(prevBest);
308         tBuffer.Enqueue(t);
309
```

```
310         // CV, SS 2021-5-27: Check if fixed queue size is working using tBuffer
311         // by output buffer on console
312         /*string deltaDistString = "";
313         foreach(float n in tBuffer)
314         {
315             deltaDistString = deltaDistString + ", " + n;
316         }
317         Debug.LogWarning(deltaDistString);*/
318
319         /*using(StreamWriter writer = new StreamWriter(path,true)) {
320             writer.WriteLine("Ground");
321             writer.Close();
322         }*/
323
324
325         if(reward_flag == true) // CV 2021-5-19: reward function with delta_R
326         {
327             if (distance > prevBest) // penalty (positve x-axis)
328             {
329                 reward = -1*k1*delta_distance; // CV 2021-5-19: Line 1
330             }
331             else // reward (negative x-axis)
332             {
333                 reward = k2*(delta_R-delta_distance); // CV 2021-5-19: Line 2
334                 prevBest = distance;
335             }
336         }
337         else // CV 2021-5-14: Original algorithm (slightly modidfied to add
                reward outside of if statement)
338         {
339             if (distance > prevBest)
340             {
341                 // Penalty if the arm moves away from the closest position to
                        target
342                 reward = prevBest - distance;
343             }
```

```
344             else
345             {
346                 // Reward if the arm moves closer to target
347                 reward = diff;
348                 prevBest = distance;
349             }
350         }
351 
352         AddReward(reward);  // CV 2021-5-14: Adds reward given by the cases
                above
353 
354         // CV 2021-5-14: Wrtie to CSV file
355         //    1) t = time index (CV 2021-5-17)
356         //    2) beginDistance = initial distance between end effector and
                target (fixed
357         //       for each episode)
358         //    3) prevBest = closest distance between end effector and target
                saved
359         //    4) distance = distance between current end effector and target
360         //    5) reward
361         //    6) nearestComponent's position // CV,HL,SS 2021-6-2
362         //    6) endEffector's position // CV,HL,SS 2021-6-2
363 
364         using(StreamWriter writer = new StreamWriter(path,true)) {
365             writer.WriteLine( t + "," +
366                               beginDistance + "," +
367                               prevBest + "," +
368                               distance + "," +
369                               delta_distance + "," +
370                               reward + "," +
371                               nearestComponent.transform.position.x + "," +
372                               nearestComponent.transform.position.y + "," +
373                               nearestComponent.transform.position.z + "," +
374                               endEffector.transform.position.x + "," +
375                               endEffector.transform.position.y + "," +
376                               endEffector.transform.position.z );
```

```
377                 writer.Close();
378             }
379             t++; // CV 2021-5-17: increments  the time index
380 
381             AddReward(stepPenalty);
382         }
383     }
384 
385     //--------------------------------------------------------------/
386     // GroundHitPenalty()
387     //    - Original author's function to give hefty penalty if robot
388     //      arm or gripper collides with ground
389     //    - called in PenaltyColliders.cs file in same folder
390 
391     public void GroundHitPenalty()
392     {
393         AddReward(-1f);
394 
395         // CV 2021-5-17: Stores "Ground" in CSV file if ground hit
396         /*using(StreamWriter writer = new StreamWriter(path,true)) {
397             writer.WriteLine("Ground");
398             writer.Close();
399         }*/
400 
401         EndEpisode();
402     }
403 
404     //--------------------------------------------------------------/
405     // Monobehavior::OnTriggerEnter()
406     //    - Unity's function where this function is called when a GameObject
407     //      collides with another GameObject
408     //    - parameters:
409     //        + other = the other Collider involved in the collision
410     //    - Unity API reference: https://docs.unity3d.com/ScriptReference/
        Collider.OnTriggerEnter.html
411 
```

```
412     private void OnTriggerEnter(Collider other)
413     {
414         JackpotReward(other);
415 
416         /*// CV 2021-5-17: Stores "Jackpot" in CSV file if end effector reached
                target
417         using(StreamWriter writer = new StreamWriter(path,true)) {
418             writer.WriteLine("Jackpot");
419             writer.Close();
420         }*/
421     }
422 
423     //-----------------------------------------------------------/
424     // JackpotReward()
425     //   - add rewards when end effector reaches the target, where
426     //           reward = 0.5 + bonus
427     //           bonus = target (dot) endEffector
428     //   - Notes: from https://docs.unity3d.com/ScriptReference/Vector3.Dot.html
429     //       + dot product of two vectors (x and y are vectors)
430     //               x (dot) y = ||x|| ||y|| cos(theta)
431     //       + for normalized vectors, it will return
432     //            1)  1 = point in same direction
433     //            2) -1 = point in completely opposite directions
434     //            3)  0 = vectors are perpendicular
435 
436     public void JackpotReward(Collider other)
437     {
438         if (other.transform.CompareTag("Components"))
439         {
440             float SuccessReward = 0.5f;
441             float bonus = Mathf.Clamp01(Vector3.Dot(nearestComponent.transform.up.
                    normalized,
442                                 endEffector.transform.up.normalized));
443             float reward = SuccessReward + bonus;
444             if (float.IsInfinity(reward) || float.IsNaN(reward)) return;
```

```
445             Debug.LogWarning("Great! Component reached. Positive reward:" +
                    reward );
446             AddReward(reward);
447             //EndEpisode();
448             UpdateNearestComponent();
449         }
450     }
451
452     //------------------------------------------------------------/
453     // private float[] NormalizedAngles()
454     // {
455     //     float[] normalized = new float[6];
456     //     for (int i = 0; i < 6; i++)
457     //     {
458     //         normalized[i] = angles[i] / 360f;
459     //     }
460     //
461     //     return normalized;
462     // }
463
464     //------------------------------------------------------------/
465     // MoveToSafeRandomPosition()
466     //   - Original author's function to make sure that endEffector is
467     //     above the ground but not out of reach
468     //   - Function called in
469     //       1) Initialize()
470     //       2) OnEpisodeBegin()
471
472     private void MoveToSafeRandomPosition()
473     {
474         int maxTries = 100;
475
476         while (maxTries > 0)
477         {
478             armAxes.All(axis =>
479                 {
```

```
480                 Axis ax = axis.GetComponent<Axis>();
481                 Vector3 angle = ax.rotationAxis * Random.Range(ax.MinAngle, ax.
                        MaxAngle);
482                 ax.transform.localRotation = Quaternion.Euler(angle.x, angle.y,
                        angle.z);
483                 return true;
484             }
485         );
486         Vector3 tipPosition = endEffector.transform.TransformPoint(Vector3.
                zero);
487         Plane groundPlane = new Plane(Vector3.up, Vector3.zero);
488         float distanceFromGround = groundPlane.GetDistanceToPoint(tipPosition)
                ;
489         if (distanceFromGround > 0.1f && distanceFromGround <= 1f &&
                tipPosition.y > 0.01f)
490         {
491             break;
492         }
493         maxTries--;
494     }
495 }
496
497 private void MoveToSafeUserPreDefinedPostion()
498 {
499     int maxTries = 100;
500
501     while (maxTries > 0)
502     {
503         armAxes.All(axis =>
504             {
505                 Axis ax = axis.GetComponent<Axis>();
506                 // 2021-6-8: fixed position
507                 //Vector3 angle = ax.rotationAxis * Random.Range(ax.MinAngle, ax
                        .MaxAngle);
508                 Vector3 angle = ax.rotationAxis * (1.1f*ax.MinAngle+0.9f*ax.
                        MaxAngle)/2;
```

```
509
510                 ax.transform.localRotation = Quaternion.Euler(angle.x, angle.y,
                        angle.z);
511                 return true;
512             }
513         );
514         Vector3 tipPosition = endEffector.transform.TransformPoint(Vector3.
                zero);
515         Plane groundPlane = new Plane(Vector3.up, Vector3.zero);
516         float distanceFromGround = groundPlane.GetDistanceToPoint(tipPosition)
                ;
517         if (distanceFromGround > 0.1f && distanceFromGround <= 1f &&
                tipPosition.y > 0.01f)
518         {
519             break;
520         }
521         maxTries--;
522     }
523 }
524
525 //-------------------------------------------------------------/
526 // Monobehavior::Update()
527 //   - Unity's function that is called in very frame
528 //   - Specifically, this draws a green line between the
529 //     endeffector and targer
530 //   - Unity API reference: https://docs.unity3d.com/ScriptReference/
        MonoBehaviour.Update.html
531
532 private void Update()
533 {
534     if(nearestComponent != null)
535         Debug.DrawLine(endEffector.transform.position,nearestComponent.
                transform.position, Color.green);
536 }
537 }
538
```

```
539 //---------------------- Random Basic Notes ----------------------/
540 // MonoBehaviour Class
541 //   - the base calss from which every Unity script derives
542 //   - provides a framework which allows you to attach script to GameObject
543 //     in the editor
544 //   - reference:
545 //       + https://docs.unity3d.com/ScriptReference/MonoBehaviour.html
546 //       + https://docs.unity3d.com/Manual/class-MonoBehaviour.html
547 //
548 // Agent Class
549 //   - an agent is an actor that can
550 //       1) observe environment
551 //       2) decide best course of action using observation
552 //       3) execute those actions within environmnet
553 //   - agents in environment operate on "steps"
554 //   - at each steps, agent does
555 //       1) collects observation
556 //       2) passes them to decision-making policy
557 //       3) receives an action in response
558 //   - assign decision-making policy to an agent with "BehaviorParameters"
559 //     component attached to agent's GameObject
560 //   - to trigger agent decision automatically, attach "DecisionRequester"
561 //     component to agent GameObject
562 //       + "DecisionPeriod" = the frequency with which the agent requests a
        decision
563 //           Ex: DecisioPeriod of 5 means that Agent will requiest a decision
        very 5
564 //               Academy steps
565 //       + "TakeActionsBetweenDecisions" = indicates if agent should take
        action
566 //             during the Academy steps where it does not request a decision
567 //   - reference:
568 // https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.
        Agent.html#Unity_MLAgents_Agent_OnActionReceived_System_Single___
```

## Appendix C

### MODIFIED REWARD FUNCTION

This appendix shows the algorithm code for the modified reward function. The code is similar to the Baseline algorithm in the appendix B [43]. The code should be in the same file structure. The difference comes in line number 293. Here, we assigned *True* Boolean value to the *reward_flag*. Therefore, the modified algorithm will execute from line numbers 326 to 353. Please check the code below for more information and understanding.

```
1  //-------------------------------------------------------------/
2  // program     : RobotControllerAgent_piece-chee-2021-5-14.cv  /
3  // coded by    : Unity                                         /
4  // updated by : Chee Vang, HL                                  /
5  // note        : 1. re-wrote reward algorithm as picewise with /
6  //                  two line segments & k1,k2 gain factor      /
7  //-------------------------------------------------------------/
8  using System;
9  using System.Data.Common;
10 using System.Linq;
11 using Unity.MLAgents;
12 using Unity.MLAgents.Sensors;
13 using UnityEngine;
14 using Random = UnityEngine.Random;
15 // 2021-5-14: File I/O
16 using System.IO;
17 using System.Text;
18 // 2021-5-17: List/Queues
19 using System.Collections.Generic;
20
21 //-------------------------------------------------------------/
22 public class RobotControllerAgent : Agent
23 {
24     [SerializeField]
25     private GameObject[] armAxes;
```

```
26      [SerializeField]
27      private GameObject endEffector;
28
29      public bool trainingMode;
30      private bool inFrontOfComponent = false;
31      public GameObject nearestComponent;
32      private Ray rayToTest = new Ray();
33      private float[] angles = new float[5];
34      private KinematicsCalculator calculator;
35      private float beginDistance;
36      private float prevBest;
37      bool  inrange2 = false, inrange3 = false, inrange4 = false, inrange5 = false
            ;
38      private float baseAngle;
39      private const float stepPenalty = -0.0001f;
40      private string path = @"/home/chee/Documents/RobotArmMLAgentUnity/distance.
            csv";
41      // CV 2021-5-14: File path for CSV file
42
43      private int epi_index = 1;
44      // CV 2021-5-14: index used to determine number of times OnEpisodeBegin was
            called
45      private int t;
46      // CV 2021-5-17: a time index
47      private int buffLength = 3;
48      //1024;
49      // Queue
50      private Queue<float> deltaDistBuffer = new Queue<float>();
51      // CV 2021-5-20: Buffer for delta_distance
52      private Queue<float> distanceBuffer = new Queue<float>();
53      // CV 2021-5-20: Buffer for distance
54      private Queue<float> prevBestBuffer = new Queue<float>();
55      // CV 2021-5-20: Buffer for prevBest
56      private Queue<float> tBuffer = new Queue<float>();
57      // CV 2021-5-20: Buffer for time index t
58
```

```
59      //------------------------------------------------------------/
60      // CV MonoBehaviour::Start()
61      // called on frame when script is enabled just before any update methods
62      //      are called the first time
63      //- referece: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.
            html
64
65
66      private void Start()
67      {
68
69      }
70
71      //------------------------------------------------------------/
72      // Agent::Initialize ()
73      //   - Perfroms a one-time initialization or set up of the agent instance
74      //   - MLAgents API referece:
75      //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.
            MLAgents.Agent.html#Unity_MLAgents_Agent_Initialize
76
77
78      public override void Initialize()
79      {
80          ResetAllAxis();
81          // 2021-6-8: fixed starting endeffector position
82          MoveToSafeUserPreDefinedPostion();
83          //MoveToSafeRandomPosition();
84          if (!trainingMode) MaxStep = 0;
85          // CV, SS 2021-5-26: Label the columns in CSV file
86
87          using(StreamWriter writer = new StreamWriter(path,false)) {
88              writer.WriteLine("t" + "," +
89                    "beginDistance" + "," +
90                    "prevBest" + "," +
91                    "distance" + "," +
92                    "delta_distance" + "," +
```

```
93                  "reward" + "," +
94                  "target x" + "," +
95                  "target y" + "," +
96                  "target z" + "," +
97                  "endEff x " + "," +
98                  "endEff y " + "," +
99                  "endEff z ");
100             writer.Close();
101         }
102
103     }
104
105     //-----------------------------------------------------------/
106     // ResetAllAsix()
107     //   - resets all axis to zero
108     //   - function called in
109     //         1) Initialized()
110     //         2) OnEpisodeBegin()
111
112     private void ResetAllAxis()
113     {
114         armAxes.All(c =>
115         {
116             c.transform.localRotation =  Quaternion.Euler(0f, 0f, 0f);
117             return true;
118         });
119     }
120
121     //-----------------------------------------------------------/
122     // Agent::OnEpisodeBegin()
123     //   - Set up an agent instance to the start of an episode
124     //   - Resets agent and environment to their starting position where
125     //     reset values are randomized
126     //   - MLAgents API reference:
127     //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.
            MLAgents.Agent.html#Unity_MLAgents_Agent_OnEpisodeBegin
```

```
128     //    - MLAgents Github reference:
129     //https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-
            Environment-Design-Agents.md
130
131     public override void OnEpisodeBegin()
132     {
133         if(trainingMode)
134             ResetAllAxis();
135
136         // 2021-6-8: fixed starting endeffector position
137         MoveToSafeUserPreDefinedPostion();
138         //MoveToSafeRandomPosition();
139         UpdateNearestComponent();
140
141         // CV 2021-5-17: Resets time index at start of new episode
142         //    - time_index is updated at each step and resetted at the
143         //      start of a new episode
144
145         t = 0;
146
147         // CV 2021-5-14: Notifies start of a new episode in the CSV file
148         //    - epi_index = number of times this OnEpisdoeBegin() is called
149         //    - CompletedEpisodes = MLAgents C# property that returns number
150         //      of completed episodes
151         //     MLAgents API reference:
152         //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.
                MLAgents.Agent.html
153         /*using(StreamWriter writer = new StreamWriter(path,true)) {
154        //writer.WriteLine("--- Start New Episode #" + epi_index + " --- Agent Ep:
                " + CompletedEpisodes);
155             writer.Close();
156         }
157         epi_index++;*/
158
159         // CV 2021-6-2: Prints the initial data into CSV file
160
```

```
161         float distance = Vector3.Distance(endEffector.transform.TransformPoint(
                Vector3.zero),
162             nearestComponent.transform.position);
163         float delta_distance = distance - prevBest; // CV 2021-5-19: delta
                distance
164         float reward = 0;
165         using(StreamWriter writer = new StreamWriter(path,true)) {
166             writer.WriteLine( t + "," +
167                               beginDistance + "," +
168                               prevBest + "," +
169                               distance + "," +
170                               delta_distance + "," +
171                               reward + "," +
172                               nearestComponent.transform.position.x + "," +
173                               nearestComponent.transform.position.y + "," +
174                               nearestComponent.transform.position.z + "," +
175                               endEffector.transform.position.x + "," +
176                               endEffector.transform.position.y + "," +
177                               endEffector.transform.position.z );
178             writer.Close();
179         }
180         t++;
181     }
182
183     //-------------------------------------------------------------/
184     // UpdateNearestComponent()
185     //   - Original author's function to update the target ("nearestComponent")
186     //     position to a random, reachable location
187     //   - function is called when
188     //       1) OnEpisodeBegin() is called
189     //       2) end effector reaches the target (Note: orginal code does
190     //          not end episode when end effector reaches target, see
191     //          JackpotRewar() for more details)
192     //   - computes "beginDistance" and initializes "prevBest" with that value
193     //   - checks that base angle (the whole robot arm, in Unity it is the
            prefab
```

```
194     //      "RobotwithColliders" in the hierarchy) is not negative value
195     //        ---> will need to confirm with Shifa
196
197
198     private void UpdateNearestComponent()
199     {
200         /*if (trainingMode)
201         {
202             inFrontOfComponent = UnityEngine.Random.value > 0.5f;
203         }
204         if(!inFrontOfComponent)
205             nearestComponent.transform.position = transform.position +
206 new Vector3(Random.Range(0.3f,0.6f),Random.Range(0.1f,0.3f), Random.Range(0.3f
        ,0.6f));
207         else
208         {
209             nearestComponent.transform.position = endEffector.transform.
                    TransformPoint(Vector3.zero) +
210  new Vector3(Random.Range(0.01f,0.15f),Random.Range(0.01f,0.15f), Random.Range
        (0.01f,0.15f));
211         }*/
212
213         // CV 2021-5-27: fixed target location
214
215         nearestComponent.transform.position = transform.position + new Vector3
                (0.45f,0.2f,0.45f);
216
217         beginDistance = Vector3.Distance(endEffector.transform.TransformPoint(
                Vector3.zero),
218  nearestComponent.transform.position);
219         prevBest = beginDistance;
220
221         baseAngle = Mathf.Atan2( transform.position.x - nearestComponent.
                transform.position.x,
222  transform.position.z - nearestComponent.transform.position.z) * Mathf.Rad2Deg;
223         if (baseAngle < 0) baseAngle = baseAngle + 360f;
```

```
224     }
225
226     //------------------------------------------------------------/
227     // Agent::CollectObservations()
228     //   - MLAgents C# function to collect vector observations of the agents for
            the step
229     //     that describes the current environment from the perspective of the
            agent
230     //   - is called every step when agent requests a decision
231     //   - MLAgents API reference:
232
233     //   - MLAgents Github reference:
234     //https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-
            Environment-Design-Agents.md
235     /// <summary>
236     /// Markov Decision Process - Observes state for the current time step
237     /// </summary>
238     /// <param name="sensor"></param>
239
240
241     public override void CollectObservations(VectorSensor sensor)
242     {
243         sensor.AddObservation(angles);
244         sensor.AddObservation(transform.position.normalized);
245         sensor.AddObservation(nearestComponent.transform.position.normalized);
246         sensor.AddObservation(endEffector.transform.TransformPoint(Vector3.zero).
                normalized);
247         Vector3 toComponent = (nearestComponent.transform.position - endEffector.
                transform.TransformPoint(Vector3.zero));
248         sensor.AddObservation(toComponent.normalized);
249         sensor.AddObservation(Vector3.Distance(nearestComponent.transform.
                position,endEffector.transform.TransformPoint(Vector3.zero)));
250         sensor.AddObservation(StepCount / 5000);
251     }
252
253     //------------------------------------------------------------/
```

```
254     // Agent::OnActionReceived()
255     //   - MLAgents C# function used to specify the agent behaviour at every
256     //     step based on the provided action
257     //   - is called every time the agent receives an action to take
258     //   - common to assign rewards in here
259     //   - MLAgents API reference:
260     //https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.
            MLAgents.Agent.html#
            Unity_MLAgents_Agent_OnActionReceived_System_Single___
261     //   - MLAgents Github reference:
262     //https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-
            Environment-Design-Agents.md
263
264
265     public override void OnActionReceived(float[] vectorAction)
266     {
267         angles = vectorAction;
268         if (trainingMode)
269         {
270             // Translate the floating point actions into Degrees of rotation for
                    each axis
271
272             armAxes[0].transform.localRotation =
273                 Quaternion.AngleAxis(angles[0] * 180f, armAxes[0].GetComponent<Axis
                        >().rotationAxis);
274             armAxes[1].transform.localRotation =
275                 Quaternion.AngleAxis(angles[1] * 90f, armAxes[1].GetComponent<Axis
                        >().rotationAxis);
276             armAxes[2].transform.localRotation =
277                 Quaternion.AngleAxis(angles[2] * 180f, armAxes[2].GetComponent<Axis
                        >().rotationAxis);
278             armAxes[3].transform.localRotation =
279                 Quaternion.AngleAxis(angles[3] * 90f, armAxes[3].GetComponent<Axis
                        >().rotationAxis);
280             armAxes[4].transform.localRotation =
```

```
281             Quaternion.AngleAxis(angles[4] * 90f, armAxes[4].GetComponent<Axis
                    >().rotationAxis);
282
283         float distance = Vector3.Distance(endEffector.transform.TransformPoint
                (Vector3.zero),
284             nearestComponent.transform.position);
285         float diff = beginDistance - distance;
286         float delta_distance = distance - prevBest; // CV 2021-5-19: delta
                distance
287         float delta_R = beginDistance-prevBest; // CV 2021-5-19:
288
289         float k1 = 1.0f;
290     // 2021-5-14 CV,HL: gain factor for Line 1
291         float k2 = 1.5f;
292         // 2021-5-14 CV,HL: gain factor for Line 2
293         bool reward_flag = true;
294     // 2021-5-17 CV: flag for using original or our reward function
295         float reward;
296
297         // CV,SS 2021-5-27: updates the queues with latest 1024 data
298
299         if(deltaDistBuffer.Count >= buffLength)
300         {
301             deltaDistBuffer.Dequeue();
302             distanceBuffer.Dequeue();
303             prevBestBuffer.Dequeue();
304             tBuffer.Dequeue();
305         }
306         deltaDistBuffer.Enqueue(delta_distance);
307         distanceBuffer.Enqueue(distance);
308         prevBestBuffer.Enqueue(prevBest);
309         tBuffer.Enqueue(t);
310
311         // CV,SS 2021-5-27: Check if fixed queue size is working using tBuffer
312         // by output buffer on console
313         /*string deltaDistString = "";
```

```
314             foreach(float n in tBuffer)
315             {
316                 deltaDistString = deltaDistString + ", " + n;
317             }
318             Debug.LogWarning(deltaDistString);*/
319
320             /*using(StreamWriter writer = new StreamWriter(path,true)) {
321                 writer.WriteLine("Ground");
322                 writer.Close();
323             }*/
324
325
326             if(reward_flag == true) // CV 2021-5-19: reward function with delta_R
327             {
328                 if (distance > prevBest) // penalty (positve x-axis)
329                 {
330                     reward = -1*k1*delta_distance; // CV 2021-5-19: Line 1
331                 }
332                 else // reward (negative x-axis)
333                 {
334                     reward = k2*(delta_R-delta_distance); // CV 2021-5-19: Line 2
335                     prevBest = distance;
336                 }
337             }
338             else // CV 2021-5-14: Original algorithm (slightly modidfied to add
                     reward outside of if statement)
339             {
340                 if (distance > prevBest)
341                 {
342                     // Penalty if the arm moves away from the closest position to
                            target
343                     reward = prevBest - distance;
344                 }
345                 else
346                 {
347                     // Reward if the arm moves closer to target
```

```
348                 reward = diff;
349                 prevBest = distance;
350             }
351         }
352 
353         AddReward(reward);  // CV 2021-5-14: Adds reward given by the cases
                above
354 
355         // CV 2021-5-14: Wrtie to CSV file
356         //    1) t = time index (CV 2021-5-17)
357         //    2) beginDistance = initial distance between end effector and
                target (fixed
358         //       for each episode)
359         //    3) prevBest = closest distance between end effector and target
                saved
360         //    4) distance = distance between current end effector and target
361         //    5) reward
362         //    6) nearestComponent's position // CV,HL,SS 2021-6-2
363         //    6) endEffector's position // CV,HL,SS 2021-6-2
364 
365         using(StreamWriter writer = new StreamWriter(path,true)) {
366             writer.WriteLine( t + "," +
367                               beginDistance + "," +
368                               prevBest + "," +
369                               distance + "," +
370                               delta_distance + "," +
371                               reward + "," +
372                               nearestComponent.transform.position.x + "," +
373                               nearestComponent.transform.position.y + "," +
374                               nearestComponent.transform.position.z + "," +
375                               endEffector.transform.position.x + "," +
376                               endEffector.transform.position.y + "," +
377                               endEffector.transform.position.z );
378             writer.Close();
379         }
380         t++; // CV 2021-5-17: increments  the time index
```

```
381
382             AddReward(stepPenalty);
383         }
384     }
385
386     //--------------------------------------------------------------/
387     // GroundHitPenalty()
388     //   - Original author's function to give hefty penalty if robot
389     //     arm or gripper collides with ground
390     //   - called in PenaltyColliders.cs file in same folder
391
392     public void GroundHitPenalty()
393     {
394         AddReward(-1f);
395
396         // CV 2021-5-17: Stores "Ground" in CSV file if ground hit
397         /*using(StreamWriter writer = new StreamWriter(path,true)) {
398             writer.WriteLine("Ground");
399             writer.Close();
400         }*/
401
402         EndEpisode();
403     }
404
405     //--------------------------------------------------------------/
406     // Monobehavior::OnTriggerEnter()
407     //   - Unity's function where this function is called when a GameObject
408     //     collides with another GameObject
409     //   - parameters:
410     //       + other = the other Collider involved in the collision
411     //   - Unity API reference: https://docs.unity3d.com/ScriptReference/
        Collider.OnTriggerEnter.html
412
413     private void OnTriggerEnter(Collider other)
414     {
415         JackpotReward(other);
```

```
416
417         /*// CV 2021-5-17: Stores "Jackpot" in CSV file if end effector reached
                target
418         using(StreamWriter writer = new StreamWriter(path,true)) {
419             writer.WriteLine("Jackpot");
420             writer.Close();
421         }*/
422     }
423
424     //-------------------------------------------------------------/
425     // JackpotReward()
426     //    - add rewards when end effector reaches the target, where
427     //            reward = 0.5 + bonus
428     //            bonus = target (dot) endEffector
429     //    - Notes: from https://docs.unity3d.com/ScriptReference/Vector3.Dot.html
430     //          + dot product of two vectors (x and y are vectors)
431     //                x (dot) y = ||x|| ||y|| cos(theta)
432     //          + for normalized vectors, it will return
433     //              1)  1 = point in same direction
434     //              2) -1 = point in completely opposite directions
435     //              3)  0 = vectors are perpendicular
436
437     public void JackpotReward(Collider other)
438     {
439         if (other.transform.CompareTag("Components"))
440         {
441             float SuccessReward = 0.5f;
442             float bonus = Mathf.Clamp01(Vector3.Dot(nearestComponent.transform.up.
                    normalized,
443                                 endEffector.transform.up.normalized));
444             float reward = SuccessReward + bonus;
445             if (float.IsInfinity(reward) || float.IsNaN(reward)) return;
446              Debug.LogWarning("Great! Component reached. Positive reward:" +
                    reward );
447             AddReward(reward);
448             //EndEpisode();
```

```
449             UpdateNearestComponent();
450         }
451     }
452
453     //-----------------------------------------------------------/
454     // private float[] NormalizedAngles()
455     // {
456     //     float[] normalized = new float[6];
457     //     for (int i = 0; i < 6; i++)
458     //     {
459     //         normalized[i] = angles[i] / 360f;
460     //     }
461     //
462     //     return normalized;
463     // }
464
465     //-----------------------------------------------------------/
466     // MoveToSafeRandomPosition()
467     //   - Original author's function to make sure that endEffector is
468     //     above the ground but not out of reach
469     //   - Function called in
470     //         1) Initialize()
471     //         2) OnEpisodeBegin()
472
473     private void MoveToSafeRandomPosition()
474     {
475         int maxTries = 100;
476
477         while (maxTries > 0)
478         {
479             armAxes.All(axis =>
480                 {
481                     Axis ax = axis.GetComponent<Axis>();
482                     Vector3 angle = ax.rotationAxis * Random.Range(ax.MinAngle, ax.
                        MaxAngle);
```

```
483                 ax.transform.localRotation = Quaternion.Euler(angle.x, angle.y,
                        angle.z);
484                 return true;
485             }
486         );
487         Vector3 tipPosition = endEffector.transform.TransformPoint(Vector3.
                zero);
488         Plane groundPlane = new Plane(Vector3.up, Vector3.zero);
489         float distanceFromGround = groundPlane.GetDistanceToPoint(tipPosition)
                ;
490         if (distanceFromGround > 0.1f && distanceFromGround <= 1f &&
                tipPosition.y > 0.01f)
491         {
492             break;
493         }
494         maxTries--;
495     }
496 }
497
498 private void MoveToSafeUserPreDefinedPostion()
499 {
500     int maxTries = 100;
501
502     while (maxTries > 0)
503     {
504         armAxes.All(axis =>
505             {
506                 Axis ax = axis.GetComponent<Axis>();
507                 // 2021-6-8: fixed position
508                 //Vector3 angle = ax.rotationAxis * Random.Range(ax.MinAngle, ax
                        .MaxAngle);
509                 Vector3 angle = ax.rotationAxis * (1.1f*ax.MinAngle+0.9f*ax.
                        MaxAngle)/2;
510
511                 ax.transform.localRotation = Quaternion.Euler(angle.x, angle.y,
                        angle.z);
```

```
512                 return true;
513             }
514         );
515         Vector3 tipPosition = endEffector.transform.TransformPoint(Vector3.
                zero);
516         Plane groundPlane = new Plane(Vector3.up, Vector3.zero);
517         float distanceFromGround = groundPlane.GetDistanceToPoint(tipPosition)
                ;
518         if (distanceFromGround > 0.1f && distanceFromGround <= 1f &&
                tipPosition.y > 0.01f)
519         {
520             break;
521         }
522         maxTries--;
523     }
524 }
525
526 //-----------------------------------------------------------------/
527 // Monobehavior::Update()
528 //   - Unity's function that is called in very frame
529 //   - Specifically, this draws a green line between the
530 //     endeffector and targer
531 //   - Unity API reference: https://docs.unity3d.com/ScriptReference/
        MonoBehaviour.Update.html
532
533 private void Update()
534 {
535     if(nearestComponent != null)
536         Debug.DrawLine(endEffector.transform.position,nearestComponent.
                transform.position, Color.green);
537 }
538 }
539
540 //---------------------- Random Basic Notes ----------------------/
541 // MonoBehaviour Class
542 //   - the base calss from which every Unity script derives
```

```
543 //   - provides a framework which allows you to attach script to GameObject
544 //     in the editor
545 //   - reference:
546 //       + https://docs.unity3d.com/ScriptReference/MonoBehaviour.html
547 //       + https://docs.unity3d.com/Manual/class-MonoBehaviour.html
548 //
549 // Agent Class
550 //   - an agent is an actor that can
551 //       1) observe environment
552 //       2) decide best course of action using observation
553 //       3) execute those actions within environmnet
554 //   - agents in environment operate on "steps"
555 //   - at each steps, agent does
556 //       1) collects observation
557 //       2) passes them to decision-making policy
558 //       3) receives an action in response
559 //   - assign decision-making policy to an agent with "BehaviorParameters"
560 //     component attached to agent's GameObject
561 //   - to trigger agent decision automatically, attach "DecisionRequester"
562 //     component to agent GameObject
563 //       + "DecisionPeriod" = the frequency with which the agent requests a
        decision
564 //           Ex: DecisioPeriod of 5 means that Agent will requiest a decision
        very 5
565 //               Academy steps
566 //       + "TakeActionsBetweenDecisions" = indicates if agent should take
        action
567 //             during the Academy steps where it does not request a decision
568 //   - reference:
569 // https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.
        Agent.html#Unity_MLAgents_Agent_OnActionReceived_System_Single___
```

## Appendix D

### PYTHON IMPLEMENTATION FOR PERFORMANCE EVALUATION

The piece of code is the python implementation from the section 5 of experimental analysis. We use our customized plots for getting the conclusions [44]. This code requires all the 20 CSV files for reading the data. The CSV files are generated automatically when we run the scripts mentioned in appendices B and C on pages 71 and 89 respectively. We just need to give the proper location to the files in this code.

```
1  import matplotlib.pyplot as plt
2  import pandas as pd
3  #Creating a two Lists of 10 DataFrame Objects and a List of 10 Paths each for
       BaseLine and K2 Algorithms
4
5  baseLineList = [pd.DataFrame()] * 10
6  baseLinePathList = [""] * 10
7
8  k2List = [pd.DataFrame()] * 10
9  k2PathList = [""]  * 10
10
11 #Generating Paths for BaseLine and k2 Algorithms
12 for index in range(0,9):
13     baseLinePathList[index] = "~/Documents/Work/trainingData/base/distance-base
           -0"+ str(index + 1) +"-2021-6-15.csv"
14     k2PathList[index] = "~/Documents/Work/trainingData/distance/k2-1p5/distance
           -k2-1p5-0"+ str(index + 1) +"-2021-6-14.csv"
15
16 baseLinePathList[9] = "~/Documents/Work/trainingData/base/distance-base
       -10-2021-6-15.csv"
17 k2PathList[9] = "~/Documents/Work/trainingData/distance/k2-1p5/distance-k2-1p5
       -10-2021-6-14.csv"
18
19 def generateDataFrames():
20     for i in range(0,10):
21         baseLineList[i] = pd.read_csv(baseLinePathList[i],usecols = ["reward"])
```

```
22          k2List[i] = pd.read_csv(k2PathList[i],usecols = ["reward"])
23          baseLineList[i] = baseLineList[i].cumsum()
24          k2List[i] = k2List[i].cumsum()
25
26  generateDataFrames()
27
28  # Merges all dataFrames and for each unique value of index from 10 csv files,
        reward average is generated.
29  baseLine_df_avg = pd.concat(baseLineList)
30  baseLine_df_avg = baseLine_df_avg.groupby(baseLine_df_avg.index).mean()
31
32  k2_df_avg = pd.concat(k2List)
33  k2_df_avg = k2_df_avg.groupby(k2_df_avg.index).mean()
34
35  #baseLine_df_avg.to_csv('baseLineAverage.csv')
36  #k2_df_avg.to_csv('k2Average.csv')
37
38  baseLineFirstNegIndex = baseLine_df_avg[baseLine_df_avg['reward'] < 0].index[0]
39  print(f"BaseLine First Negative Index: {baseLineFirstNegIndex}")
40  baseLineCrossOverPoint = baseLine_df_avg[(baseLine_df_avg['reward'] >= 0) & (
        baseLine_df_avg.index > baseLineFirstNegIndex)].index[0]
41  print(f"BaseLine CrossOver Point: {baseLineCrossOverPoint}")
42
43  k2FirstNegIndex = k2_df_avg[k2_df_avg['reward'] < 0].index[0]
44  print(f"K2 First Negative Index: {k2FirstNegIndex}")
45  k2CrossOverPoint = k2_df_avg[(k2_df_avg['reward'] >= 0) & (k2_df_avg.index >
        k2FirstNegIndex)].index[0]
46  print(f"k2 CrossOver Point: {k2CrossOverPoint}")
47
48  print(f"Total Steps of BaseLine: {len(baseLine_df_avg.index)}")
49  print(f"Total Steps of K2: {len(k2_df_avg.index)}")
50
51  # This section is according to the Notes from Professor which computes Eta
        negative and Eta positive
52  baseLine_neg_index = baseLine_df_avg[baseLine_df_avg < 0].dropna().sum().round
        (1)
```

```
53 baseLine_pos_index = baseLine_df_avg[baseLine_df_avg > 0].dropna().sum().round
       (1)
54
55 print("BaseLine Negative Index Summation:", baseLine_neg_index.values)
56 print("BaseLine Positive Index Summation:", baseLine_pos_index.values)
57
58 k2_neg_index = k2_df_avg[k2_df_avg < 0].dropna().sum().round(1)
59 k2_pos_index = k2_df_avg[k2_df_avg > 0].dropna().sum().round(1)
60
61 print("K2 Negative Index Summation:", k2_neg_index.values)
62 print("K2 Positive Index Summation:", k2_pos_index.values)
63
64 negativeImprovement = k2_neg_index/baseLine_neg_index
65 positiveImprovement = k2_pos_index/baseLine_pos_index
66
67 print("ETA Negative:", negativeImprovement.values)
68 print("ETA Positive:", positiveImprovement.values)
69
70 #Generating DataFrames for Graphs
71 baseLine_df = pd.DataFrame({'y1':baseLineList[0]['reward'],'y2':baseLineList
       [1]['reward'],'y3':baseLineList[2]['reward'], 'y4':baseLineList[3]['reward'
       ], 'y5':baseLineList[4]['reward'],'y6':baseLineList[5]['reward'],'y7':
       baseLineList[6]['reward'],'y8':baseLineList[7]['reward'],'y9':baseLineList
       [8]['reward'],'y10':baseLineList[9]['reward'],'y11':baseLine_df_avg['reward
       ']})
72 k2_df = pd.DataFrame({'y1':k2List[0]['reward'],'y2':k2List[1]['reward'],'y3':
       k2List[2]['reward'], 'y4':k2List[3]['reward'], 'y5':k2List[4]['reward'],'y6
       ':k2List[5]['reward'],'y7':k2List[6]['reward'],'y8':k2List[7]['reward'],'y9
       ':k2List[8]['reward'],'y10':k2List[9]['reward'],'y11':k2_df_avg['reward']})
73 df_baseLine_k2 = pd.DataFrame({'BaseLine':baseLine_df_avg['reward'],'K2':
       k2_df_avg['reward']})
74
75 #Dropping rows for missing Values after merger of BaseLine and k2 DataFrame
       Objects
76 baseLine_k2_maxRange = min(baseLineMaxRange,k2MaxRange)
```

```
77  df_baseLine_k2 = df_baseLine_k2.drop(range(baseLine_k2_maxRange, len(
        df_baseLine_k2.index)))
78
79  #Scaling the graph
80  fig, axs = plt.subplots(3, figsize = (15,20), squeeze = True)
81
82  #Plotting the reward values on Y axis for each csv files
83  axs[0].set_title('BaseLine Algorithm:')
84  axs[0].set_xlabel("Steps")
85  axs[0].set_ylabel("Reward")
86  axs[0].plot('y1',data = baseLine_df, linewidth=1)
87  axs[0].plot('y2',data = baseLine_df, linewidth=1)
88  axs[0].plot('y3',data = baseLine_df, linewidth=1)
89  axs[0].plot('y4',data = baseLine_df, linewidth=1)
90  axs[0].plot('y5',data = baseLine_df, linewidth=1)
91  axs[0].plot('y6',data = baseLine_df, linewidth=1)
92  axs[0].plot('y7',data = baseLine_df, linewidth=1)
93  axs[0].plot('y8',data = baseLine_df, linewidth=1)
94  axs[0].plot('y9',data = baseLine_df, linewidth=1)
95  axs[0].plot('y10',data = baseLine_df,linewidth=1)
96
97  #Plotting the Average value of reward from all 10 csv files on Y Axis
98  axs[0].plot('y11', data = baseLine_df, color = "black", label = "Average",
        linestyle = 'solid', linewidth = '2')
99  axs[0].legend()
100
101 # Graph for K2 Algorithm
102 axs[1].set_title('K2 Algorithm:')
103 axs[1].set_xlabel("Steps")
104 axs[1].set_ylabel("Reward")
105 axs[1].plot('y1',data = k2_df, linewidth=1)
106 axs[1].plot('y2',data = k2_df, linewidth=1)
107 axs[1].plot('y3',data = k2_df, linewidth=1)
108 axs[1].plot('y4',data = k2_df, linewidth=1)
109 axs[1].plot('y5',data = k2_df, linewidth=1)
110 axs[1].plot('y6',data = k2_df, linewidth=1)
```

```
111 axs[1].plot('y7',data = k2_df, linewidth=1)
112 axs[1].plot('y8',data = k2_df, linewidth=1)
113 axs[1].plot('y9',data = k2_df, linewidth=1)
114 axs[1].plot('y10',data = k2_df,linewidth=1)
115
116 #Plotting the Average value of reward from all 10 csv files on Y Axis
117 axs[1].plot('y11', data = k2_df, color = "black", label = "Average", linestyle
        = 'solid', linewidth = '2')
118 axs[1].legend()
119
120 #Graph for Comparison between BaseLine and K2 Algorithm
121 axs[2].plot('BaseLine',label = "BaseLine",data = df_baseLine_k2, linewidth = 1,
         color = 'black')
122 axs[2].plot('K2',data = df_baseLine_k2, label = "K2", linewidth = 1, color = '
        red')
123 axs[2].plot(baseLineCrossOverPoint,baseLine_df_avg.iloc[baseLineCrossOverPoint
        ]['reward'], marker = 'o')
124 axs[2].annotate(f"baseLine Cross Over Point at ({baseLineCrossOverPoint}, {
        baseLine_df_avg.iloc[baseLineCrossOverPoint]['reward'].round(3)})", xy = (
        baseLineCrossOverPoint,baseLine_df_avg.iloc[baseLineCrossOverPoint]['reward
        ']),xytext =(baseLineCrossOverPoint + 700, -700))
125 axs[2].annotate(f"k2 Cross Over Point at ({k2CrossOverPoint}, {k2_df_avg.iloc[
        k2CrossOverPoint]['reward'].round(3)})", xy = (k2CrossOverPoint,k2_df_avg.
        iloc[k2CrossOverPoint]['reward']),xytext =(k2CrossOverPoint - 8000, 1000))
126 axs[2].plot(k2CrossOverPoint,k2_df_avg.iloc[k2CrossOverPoint]['reward'], marker
         = 'o')
127 axs[2].set_xlabel('Steps')
128 axs[2].set_ylabel('Reward')
129 axs[2].legend()
```